APPLIED ROOTS

Deep Learning

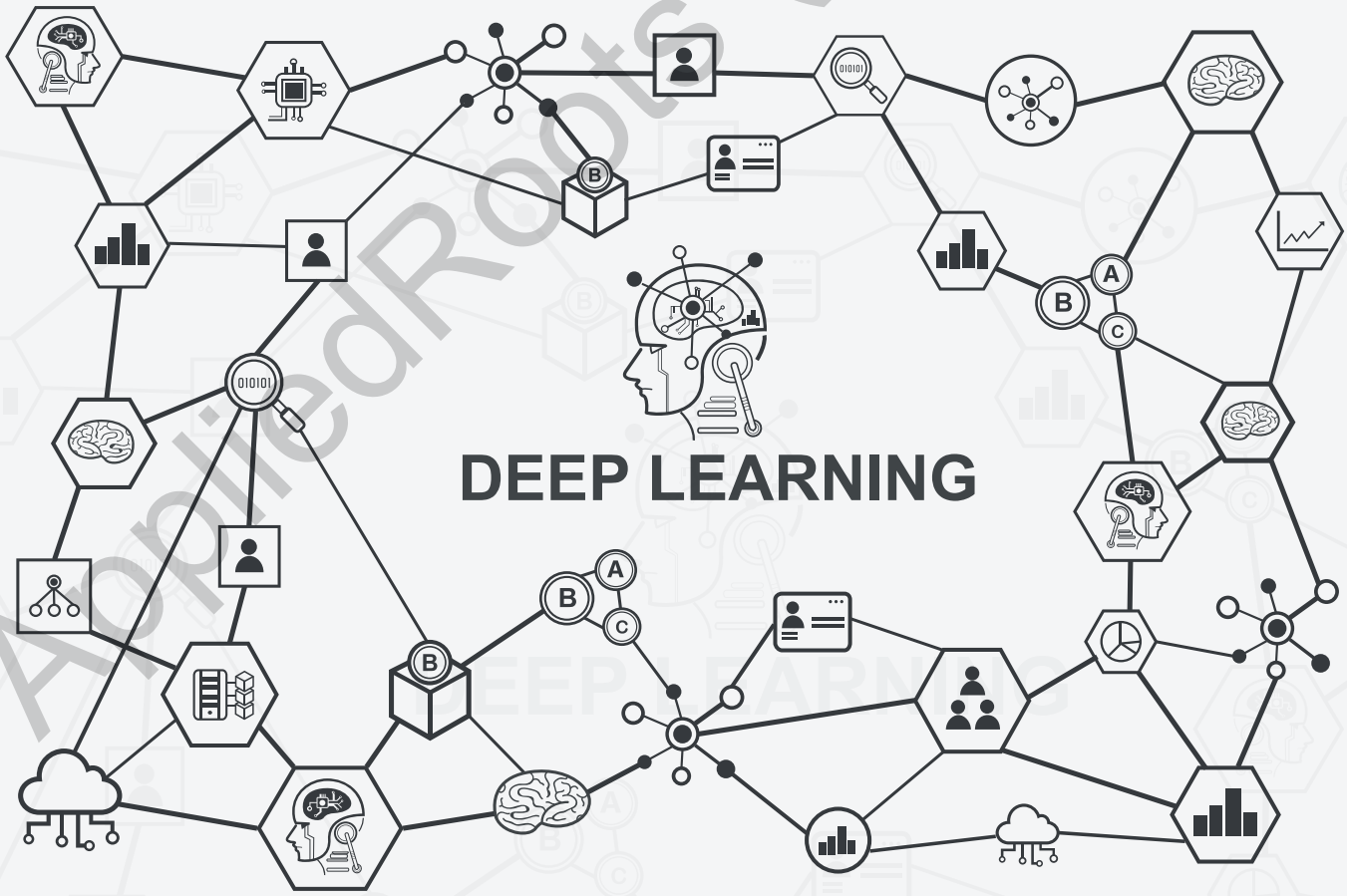


Table of Contents

01. DEEP LEARNING

DEEP LEARNING

Previously, in the chapter "Introduction to ANNs", we discussed about the basics of the neural networks and how they have the ability to learn complex nonlinear mappings between any pair of input and output variables by using stochastic gradient descent to optimize for the best weight matrix (or kernel) .

We also discussed the technique of dropout and how it prevents/reduces overfitting. Other concepts like weight initialization and advanced optimizers were also briefly touched upon.

In this chapter we shall explore these concepts further, along with some other new concepts that will help us build deeper neural networks that have even more "Learning capacity".

WEIGHT INITIALIZATION:

Since gradient descent optimization basically chooses its initial weight matrix randomly and then begins to update them based on the gradients of its loss function at those randomly initialized weights, the optimal end solution (ie: Final weight matrix) that the neural network converges to, is partially dependent on the randomly chosen initial weight matrix.

Given a complex high dimensional loss surface, suppose the randomly chosen initial weights are such that they are at some local minima or on some plateau (flat section of the loss surface), then it is quite possible that the model may take a long time to converge to a good solution or at worst it may not learn anything at all.

Also, as previously discussed in "Introduction to ANNs" the weight initialization scheme should be such that it ensures "optimal asymmetry". In other words each of the weights in the initial weight matrix should have optimal variance (ie: they should be optimally different from each other), so that each neuron learns about a different aspect of the data. Apart from asymmetry, the weight initialization scheme should be such that it ensures that the weights should neither be too large or too small.

Some of the more popular weight initialization schemes are shown below. These techniques are heuristic in nature and they are based on the number of inputs to a layer (fan in) and the number of outputs from a layer (fan out).

1. **GAUSSIAN INITIALIZATION:**

   $W \sim N(\mu = 0, \text{std\_dev} <= 0.05)$

2. **UNIFORM INITIALIZATION:**

   $W \sim \text{Uniform}(-1/\sqrt{\text{fan\_in}}, 1/\sqrt{\text{fan\_out}})$

3. **XAVIER GLOROT NORMAL INITIALIZATION:**

   $W \sim N(\mu = 0, \text{std\_dev})$ where $\text{std\_dev} = 2/\text{fan\_sum}$
   where $\text{fan\_sum} = \text{fan\_in} + \text{fan\_out}$

4. **XAVIER GLOROT UNIFORM INITIALIZATION:**

   $W \sim \text{Uniform}(\sqrt{(-6/\text{fan\_sum})}, \sqrt{(6/\text{fan\_sum})})$

5. **HE NORMAL INITIALIZATION:**

   $W \sim N(\mu = 0, \text{std\_dev})$ where $\text{std\_dev} = \sqrt{(2/(\text{fan\_in})}$

6. **HE UNIFORM INITIALIZATION:**

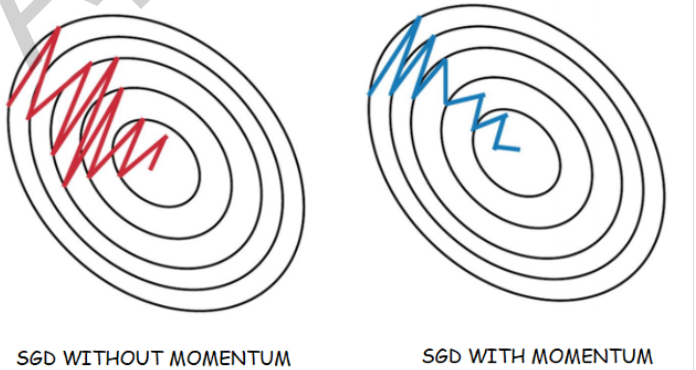
   $W \sim \text{Uniform}(\sqrt{(-6/\sqrt{\text{fan\_in}})}, \sqrt{(6/\sqrt{\text{fan\_in}})}$

ADVANCED OPTIMIZERS

1. SGD WITH MOMENTUM:

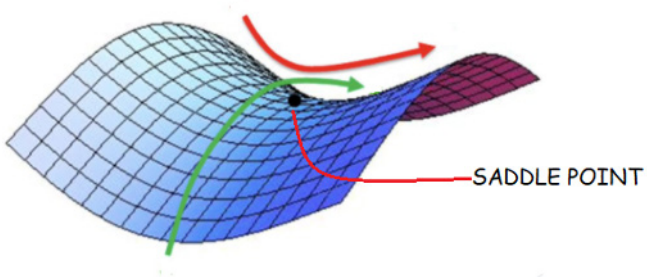
In mini batch gradient descent, the weights are updated based on gradients calculated with respect to the current batch of data points being iterated. This causes the direction of the update at every iteration to have some variance with respect to its past and so the path taken during gradient descent will not be smooth, it will instead be oscillatory all the way towards convergence.

Using momentum can reduce these oscillations. Momentum takes into account the past gradients of the model's descent path to smooth out the updates at each step of the descent. The image below expresses this phenomenon. It can be said that the collective momentum of the previous gradients makes the model take a narrower (smoother) path, thus reducing the time required for convergence.



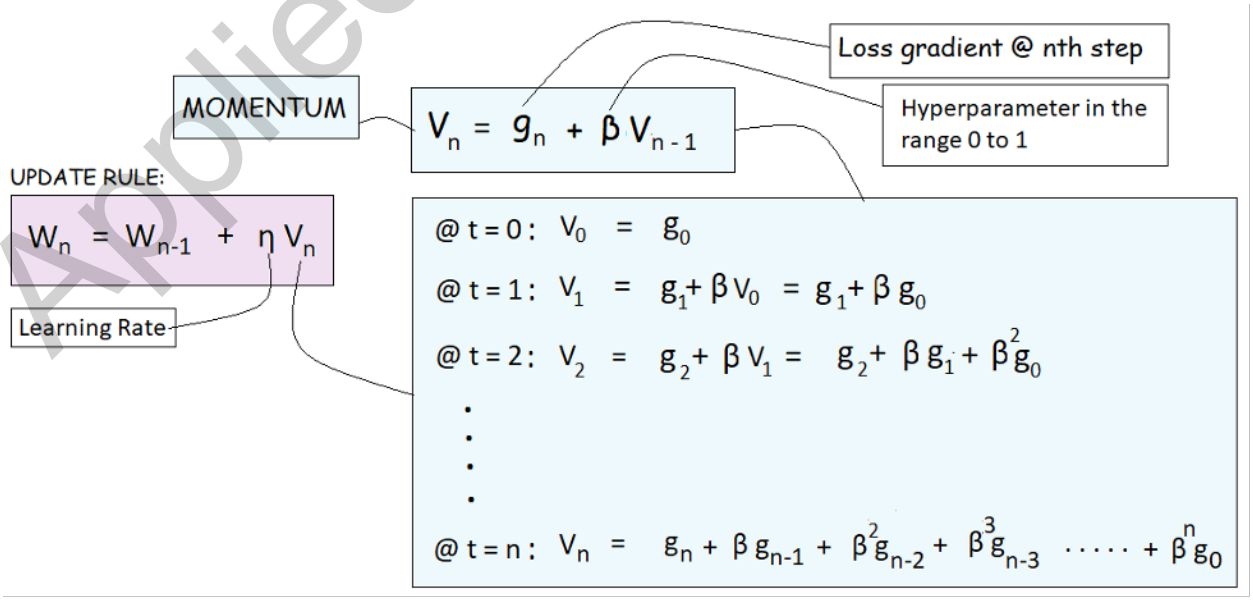
SGD WITHOUT MOMENTUM SGD WITH MOMENTUM

Apart from the above function, momentum also makes the model more resilient to local minimas and saddle points. Stochastic gradient descent computes its next step based on the gradient at the current location only and so, if the model reaches a local minima or saddle point (locations on the loss surface where it is local maxima with respect to one direction, but local maxima with respect to another), it tends to get stuck. Application of the momentum principle overcomes this problem due to the contribution of the previous gradients to the current update.



The concept of momentum is expressed in the image shown below. The momentum at the current location of the loss surface is the exponentially weighted sum of the gradients at the previous steps.

This is achieved by choosing a value between 0 to 1 for the hyperparameter β. Thus the weight associated with each previous gradient is reduced the further back it is from the current step and the current gradient has the max weight of one. The weight update is performed using the update rule shown below, where instead of using the current gradient to update the weights, the exponentially weighted sum of all the gradients till now (ie: Momentum) is used to do the same.



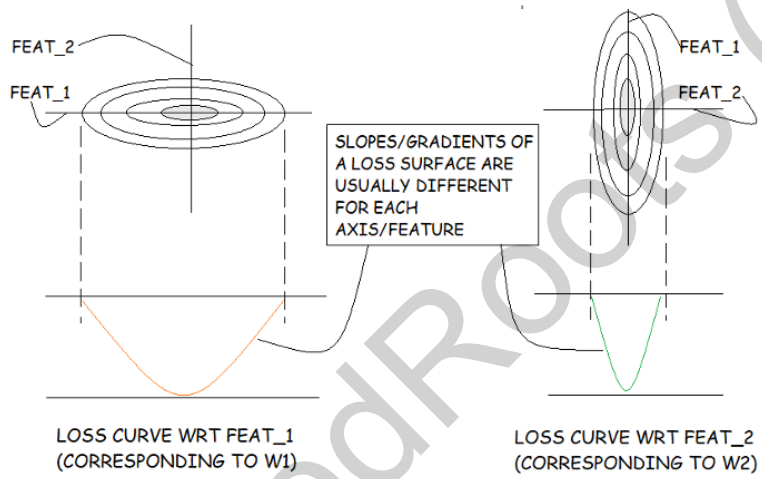
2. NESTEROV ACCELERATED GRADIENT:

Nesterov accelerated gradient is a slight variant of the SGD with momentum. Its update rule is as shown below. Here the momentum is directly added to the weights of the previous step and then this sum is updated using the current gradient multiplied by the learning rate.

$$W_n = W_{n-1} V_{n-1} + \eta g_n$$

3. ADAGRAD:

Ideally during the gradient descent towards the minima of the loss surface it would be preferable if the learning rate (η) were dynamic adjusted individually for each feature/axis based on feedback received from some metric, that takes into account the gradient information of the previous iterations. This is because - the high dimensional non convex nature of neural networks loss surfaces, could lead to different slopes/gradients with respect to each feature/axis. This is simplistically demonstrated in the image below.



Choosing the same learning rate for all features/axes may result in the learning rate being too small with respect to some particular axis and being too large with respect to another. An obvious way to mitigate this problem is to choose different learning rates for each feature. AdaGrad achieves this by adaptively scaling the learning rate with respect to the accumulated squared gradient of each iteration and it does this individually for each feature. This is expressed in the image below:

$$W_t = W_{t-1} + \Delta W$$

Original Learning Rate

$$\Delta W = \eta' g_t$$

Gradient @ current step

$$\eta' = \eta / \sqrt{(\alpha + \varepsilon)}$$

Very small value (To avoid division by zero error).

Sum of the squares of the gradients of previous iterations

Though Adagrad has the advantages of automatic tuning of the learning rates for each individual weight, there is the possibility of the alpha value in the above equation summating into large values, thus eventually leading to very small learning rates and thus very slow convergence.

4. ADA DELTA:

Adadelta is an extension of Adagrad that seeks to reduce its monotonically decreasing learning rate. Instead of inefficiently storing the sum of the squares of the previous gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. This is expressed in the image shown below:

$$W_t = W_{t-1} + \Delta W$$
$$\Delta W = \eta\, g_t$$
$$\eta = 1/(eda_t + \varepsilon)$$

Exponentially decaying average of the SQUARES of the previous gradients

$$\text{where } eda = (1-\gamma)\left(g_{t-1}^2 + \gamma\, g_{t-2}^2 + \gamma^2 g_{t-3}^2 \cdots\cdots + \gamma^n g_0^2\right)$$

Hyperparameter in the range of 0 to 1

5. ADAM:

Adam is a very effective optimizer that takes into account:

1. The exponentially decaying averages of the previous gradients.
2. The exponentially decaying averages of the squares of the previous gradients

It does this as described in the image shown below:

$$W_t = W_{t-1} + \Delta W$$
$$\Delta W = \eta \left(\frac{(eda_2 / (1-\gamma_2^2)}{\sqrt{(eda_1 / (1-\gamma_1^2) + \varepsilon}} \right)$$

where:

$$eda_1 = (1-\gamma_1)\left(g_{t-1}^2 + \gamma_1 g_{t-2}^2 + \gamma_1^2 g_{t-3}^2 \cdots\cdots + \gamma_1^n g_0^2\right)$$
$$eda_2 = (1-\gamma_2)\left(g_{t-1} + \gamma_2 g_{t-2} + \gamma_2^2 g_{t-3} \cdots\cdots + \gamma_2^n g_0\right)$$

Usually the learning rate η will be around 0.001 and eda_1 and eda_2 will be around 0.99 and 0.9 respectively. As of today Adam is the optimizer which produces the fastest convergence.

BATCH NORMALIZATION

Previously while training the Logistic Regression, we normalized the data at the pretraining stage so that the range of values in each of the features of the data were comparable (ie: They all possess a similar scale). This is a basic prerequisite for parametric models.

In the case of deep neural networks each hidden layer can be said to receive its own set of input data from the previous layer. Thus each value in the vectors outputted by a hidden layer, could be considered as belonging to some abstract feature (just like each value in the input vectors at the start of the neural network represents a feature) and they form the input vectors to the next layer.

Now, the distribution or range of values for each of these "abstract features" may not be the same. In other words, some of these abstract features might have larger scales and thus contain larger values than the rest of the features. This is because the input data to the hidden layers are actually the product of data being non linearly transformed by the previous layers. If these inputs to intermediate hidden layers are left unnormalized, the skew in distribution could get propagated to deeper layers leading to weights being learned based on skewed inputs, leading to increased chances of gradient explosion and longer training times. The technique of Batch Normalization addresses this issue.

Given a specific hidden layer, Batch normalization involves the following:

i. The output of each layer is first normalized.

$$X_{normalized} = \frac{X - mean(X)}{std\_dev(X)}$$

ii. This normalized output is further transformed using the following expression:

$$X_{input} = \gamma X_{normalized} + \beta$$

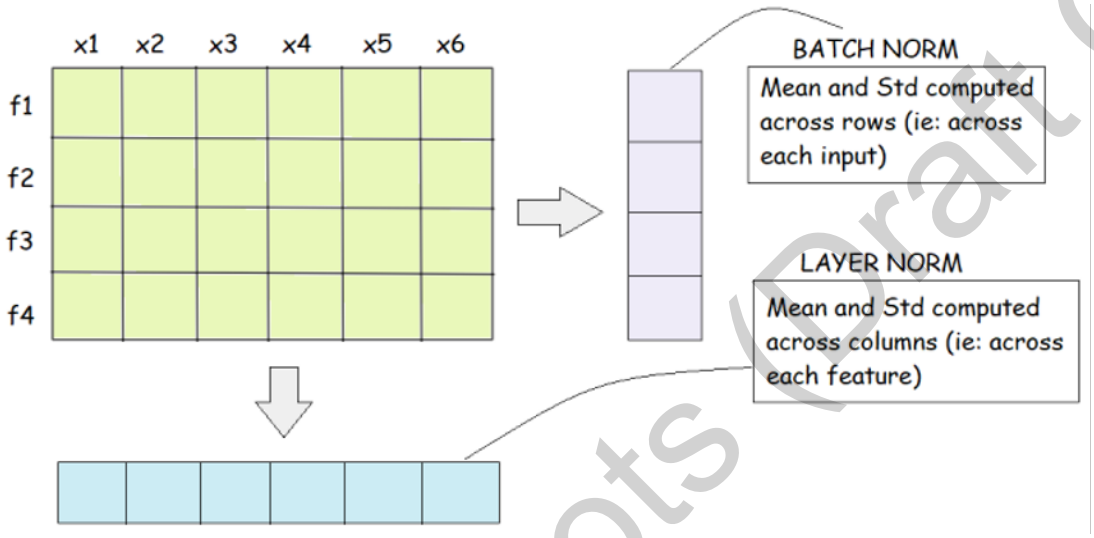
Where γ & β are arbitrary variables, whose optimal values can be learned during training (just like layer weights).

Thus we now have normalized data at the beginning of the neural network and also within the neural network itself. Batch normalization is independently applied to each batch. In other words, the parameters for normalization are learned only with respect to each batch and not across the entire training cycle.

LAYER NORMALIZATION:

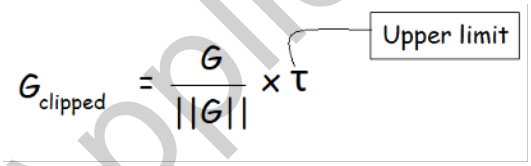
Layer normalization is another normalization technique where the same computations as shown above for batch normalization is used, but only the axis is changed. The image shown below describes how batch normalization and Layer normalization is performed with respect to a batch of data.

Batch normalization works better with fully connected layers and convolutional neural networks (CNN), whereas Layer normalization is preferred with Recurrent neural networks and transformer models as shall be seen in further chapters.



GRADIENT CLIPPING:

Gradient clipping is another technique developed to address the issue of gradient explosion. It does so by applying thresholds on the upper limits the weights can take.Consider a single hidden layer. Say the gradient vector of this layer is G = [g1, g2, g3, …. gn]. The gradient clipping would involve the operation shown in the image below:



SENTIMENT CLASSIFICATION USING DEEP LEARNING:

We shall perform sentiment classification on the TF IDF weighted w2c document vectors that we created in the chapter “Sentiment classification using word embeddings.

```
data_path = 'gdrive/My Drive/3A_PGD/CODE_PGD/NLP_SENTIMENT_CLF/'

df_tr = pd.read_csv(data_path + 'df_tr.csv')
df_ts = pd.read_csv(data_path + 'df_ts.csv')
```

We configure/design the “deep” neural network for classification as shown below. We create a 20 layer neural network with Glorot normal weight initialization. We make use of drop out layers and batch normalization layers to keep the weight matrix regularized, thus ensuring model stability fast convergence.

```
import tensorflow as tf
from tensorflow import keras

wt_init = dict(kernel_initializer = 'glorot_normal')
model_b = keras.Sequential(name = 'NN_B')

model_b.add(keras.Input(shape=(n_feats,)))
model_b.add(keras.layers.Dense(300, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(300, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(600, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(1000, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(1000, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(1000, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(5000, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(5000, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(1000, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
```

```
model_b.add(keras.layers.Dense(1000, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.5))
model_b.add(keras.layers.Dense(500, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.3))
model_b.add(keras.layers.Dense(500, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.3))
model_b.add(keras.layers.Dense(100, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.3))
model_b.add(keras.layers.Dense(100, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.3))
model_b.add(keras.layers.Dense(100, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.3))
model_b.add(keras.layers.Dense(50, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.32))
model_b.add(keras.layers.Dense(10, activation="relu", **wt_init))
model_b.add(keras.layers.BatchNormalization())
model_b.add(keras.layers.Dropout(0.3))
model_b.add(keras.layers.Dense(5,  activation="relu", **wt_init))
model_b.add(keras.layers.Dropout(0.3))
model_b.add(keras.layers.Dense(2,  activation="relu", **wt_init))

model_b.add(keras.layers.Dense(1,  activation="sigmoid", **wt_init))
```

We then train the configured model using the functions “fn_NN_binary_clf” using a batch size of 2048 over 500 epochs as shown below:

```
model = model_b
optimizer=tf.keras.optimizers.Adamax()

df_tr_, df_ts_, std_scaler = fn_standardize_df(df_tr, df_ts)
model_save_path =  data_path + 'NN_MODELS/model_b/'

batch_size, epochs = 2048, 200
class_bias = {0:87, 1:13}

z = fn_NN_binary_clf(model, optimizer, df_tr, df_ts, model_save_path,
                     batch_size=batch_size, epochs=epochs,
                     class_weight = class_bias)

df_performance_model_b, df_history_b = z
```

Increase weights of the smaller class by 1%: Give a weight ratio of 87:13 instead of 86:14

```
100%  200/200
```

We then inspect the statistical overview of the performances of the model at different epochs as shown below:

```
regd_columns = 'tr_prec_1 tr_rec_1 tr_prec_0 tr_rec_0 diff_rec_1 diff_rec_0'.split()
df_performance_model_b.loc[:, regd_columns].describe()
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
count	200.000000	200.000000	200.000000	200.000000	200.000000	200.000000
mean	0.922652	0.511805	0.304451	0.854261	0.019694	0.027865
std	0.047324	0.387438	0.128369	0.107177	0.059241	0.033387
min	0.692308	0.001206	0.140272	0.697956	0.000202	0.000443
25%	0.874240	0.007260	0.140896	0.766458	0.002897	0.004335
50%	0.950854	0.809165	0.391431	0.805843	0.008080	0.020035
75%	0.959131	0.815160	0.411576	0.992739	0.019365	0.038681
max	0.965971	0.827126	0.430376	0.998846	0.746396	0.289621

We then filter out the beast performing models as shown below:

```
dff = df_performance_model_b
df1 = dff[dff.diff_rec_0 < 0.05]
df2 = df1[df1.diff_rec_1 < 0.05]
df3 = df2[df2.tr_rec_0 > 0.815]
df_filtered = df3[df3.tr_rec_1 > 0.815]
df_filtered.loc[:, regd_columns]
```

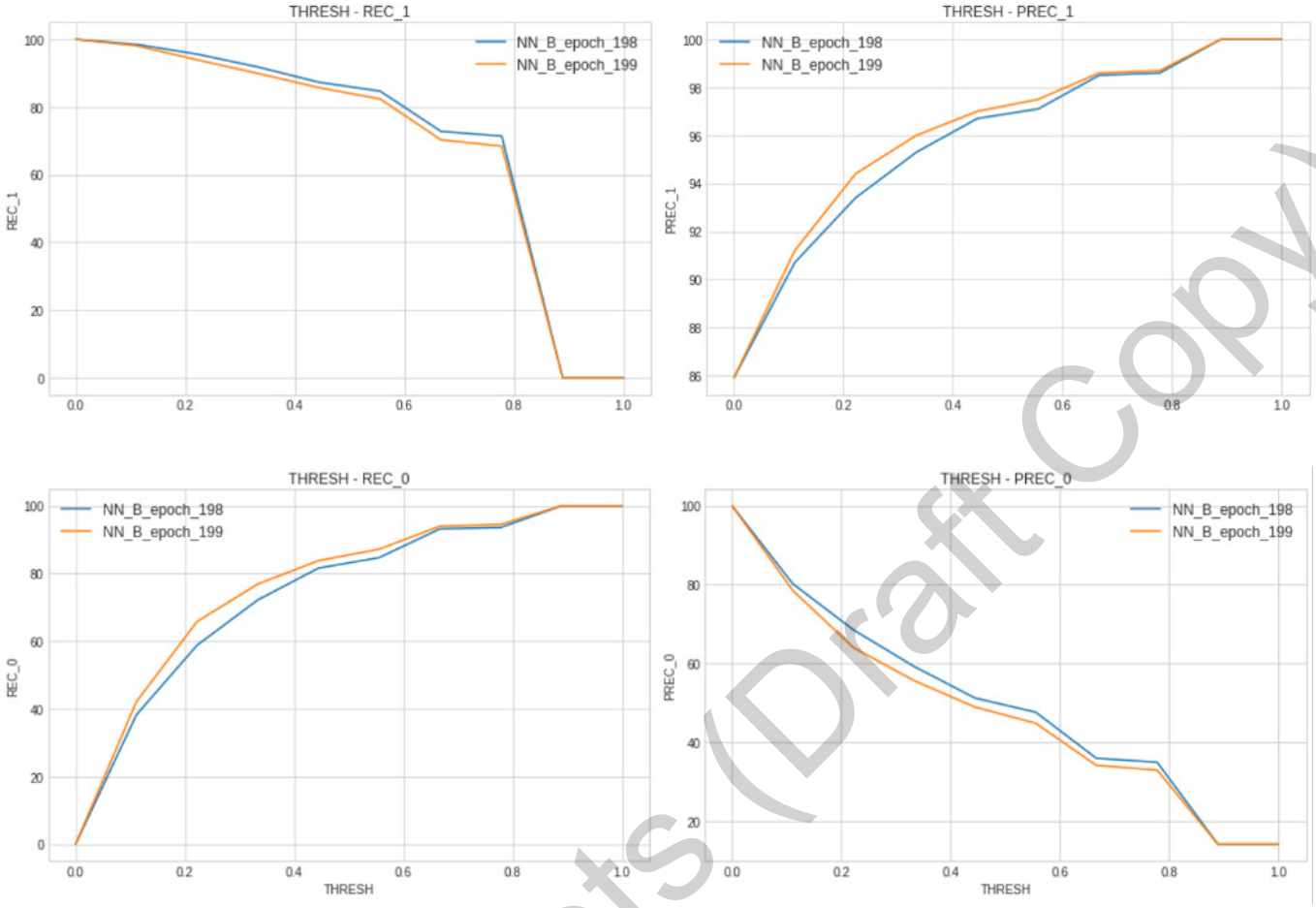
	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
NN_B_epoch_198	0.964988	0.817753	0.424174	0.818995	0.007508	0.010763
NN_B_epoch_199	0.964475	0.822633	0.429661	0.815148	0.022311	0.020397

We then check the precision recall performance of the above two models as shown below:

```
def fn_load(model_name, model_save_path = model_save_path):
    model_name = model_save_path + model_name + '.h5'
    return keras.models.load_model(model_name)
```

```
kwargs = dict(model_save_path = data_path + 'NN_MODELS/model_b/')
listO_filtered_models = [fn_load(i, **kwargs) for i in list(df_filtered.index)]
df_Xy_ = df_tr
legend = list(df_filtered.index)

%time fn_performance_models_data(listO_filtered_models, df_Xy_, legend, NN=True)
```

```
CPU times: user 2min 13s, sys: 8.67 s, total: 2min 21s
Wall time: 1min 55s
```



Since we are optimizing for recall of the negative reviews, we choose the model at epoch 199, thresholded at 0.55 for our prediction purposes. We then check for generalization of the model's performance over the train and test sets as shown below.

```
df_Xy_ = df_tr
model_ =  list0_filtered_models[1]
thresh = 0.55

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.3922
ACCURACY: 83.149
--------------------
```

	prec	rec
class_0	44.9	87.2
class_1	97.5	82.5

```
df_Xy_ = df_ts

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.4198
ACCURACY: 81.577
--------------------
```

	prec	rec
class_0	42.1	82.0
class_1	96.5	81.5

As can be seen from the outputs above, the model generalizes well over the train and test sets and it achieves a negative recall of around 82%, with around 42% precision.

02. CONVOLUTIONAL NEURAL NETWORKS

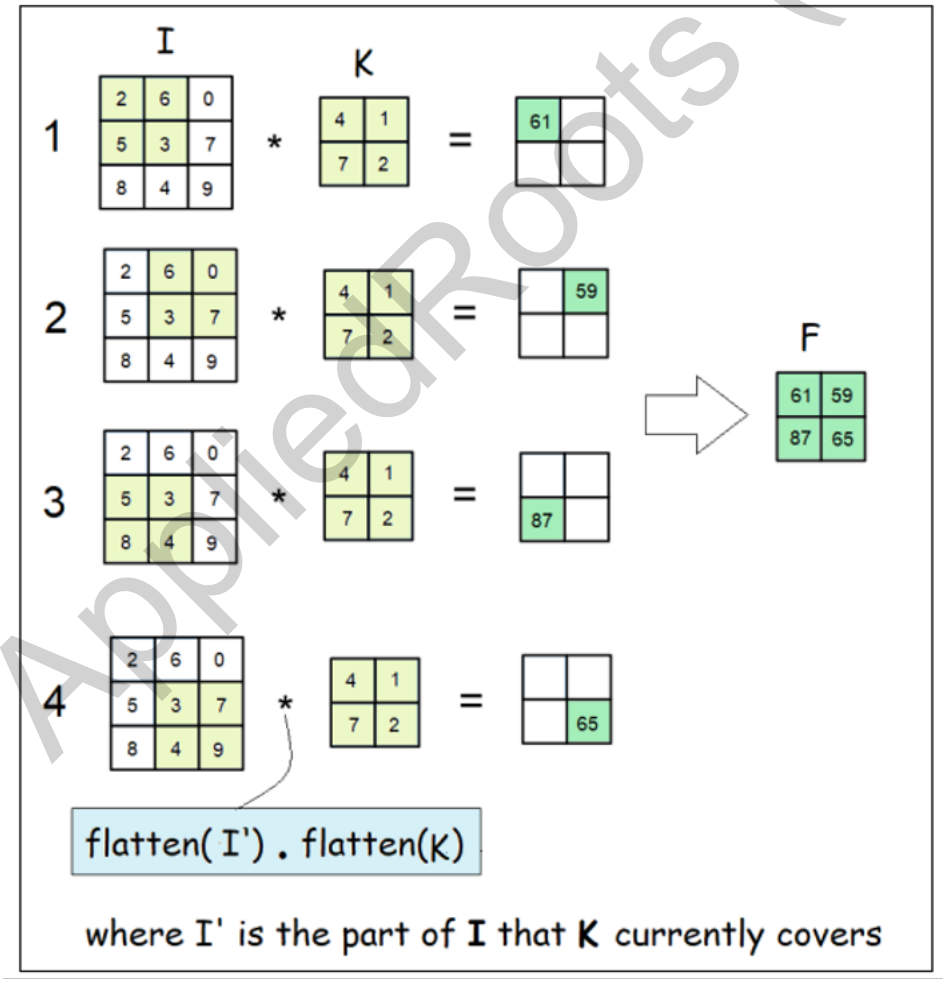
CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) refer to a category of neural network architecture which was originally designed for image analysis. CNNs contain two extra operations when compared to the neural networks discussed in the previous chapters (ie: Multi Layered Perceptrons or MLPs) namely: convolution and pooling.

Just like a MLP, a Convolutional Neural Network is made up of one input layer, multiple hidden layers, and an output layer, but the hidden layers include convolutional layers and pooling layers apart from the fully connected layers. Convolution and Pooling layers as their name suggests, perform convolution and pooling operations respectively.

THE CONVOLUTION OPERATION:

The convolution operation is the core operation in a CNN. The image below describes what in essence is considered a convolution operation.



Consider the two matrices I & K shown in the image above. K is smaller than I, and it can take four positions within matrix I. Matrix I represents the kind of input that CNNs are designed to handle (ie: Data with a grid-like topology). Images represented in matrix form, are one of the most obvious examples of this kind of data.

A convolution operation basically involves the following steps.

- In the beginning, matrix K is placed at the top left corner of I and then it is moved horizontally in equal steps.
- The step size is called "**stride**".
- Stride of 1 means K shifts over one column of matrix I per step.
- After K reaches the last position possible for its horizontal movement (K should always remain within I), it takes a step in the downward direction.
- It then repeats the same horizontal movement from the left most position as shown in step 3 in the image.
- So there are two types of strides: horizontal & vertical. In the image above both strides have a value of 1
- The process described above is repeated until the whole of matrix I has been covered.
- For every position of K on I in the sequence described above, the dot product shown in the image is computed and is used to populate another matrix F.

KERNEL:

The matrix K that moves over the input matrix I is called the **kernel**. The values within the Kernel can be viewed as **weights** that can be randomly initialized and gradually learnt via Gradient descent optimization.

Convolution operation can be performed for different specifications of stride and kernel size depending on the size of matrix I and other specific requirements of the task at hand.

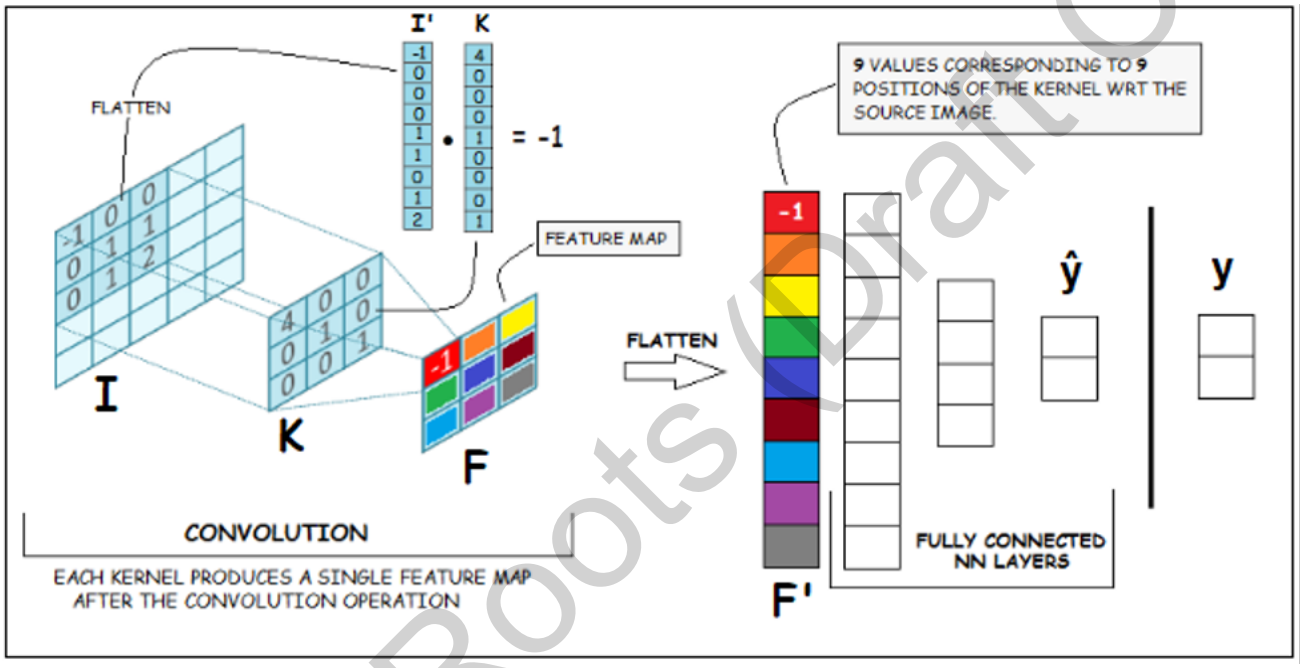
Kernels are also called **filters**, due to the fact that they can be thought of as filters through which the input image is processed.

LEARNING IN CNNS:

Just as in the case of standard neural networks (MLPs), where we learn from a dataset of 1D vectors of fixed size, in case of CNNS we learn from a dataset of images that are in the form of same sized 2D matrices.

Consider the case of image classification involving a dataset of images of two classes (say cats & dogs). The basic architecture for image classification using CNNs is shown in the image below. Here the matrix F is formed by the convolution operation of kernel K on the image inputs I. It is then flattened to form the vector F', which forms the input to a fully connected neural network. This fully connected neural network outputs a prediction $\hat{\mathbf{y}}$, which is then compared to the actual label y by using a classification loss function.

The best weights for the fully connected layers and the kernel K is then learnt via gradient descent optimization.



FEATURE MAP:

The output matrix **F**, obtained from a convolution operation is called a **Feature Map**. In the context that the Kernel matrix contains trainable weights that are optimized using gradient descent, the feature map obtained after optimal training represents information derived by "scanning" the input image matrix **I** for local information. Local information means patterns within image **I** that fit within the window size specified by the kernel's dimensions.

In other words, the kernel's weights can be trained to detect patterns anywhere within input image I, that have a size lesser than or equal to the size of the kernel. So suppose we have a 28 x 28 sized image and say we use a kernel of size 3 x 3. Then the weights of this kernel can be optimized to detect 3 x 3 sized patterns that exist within the image.

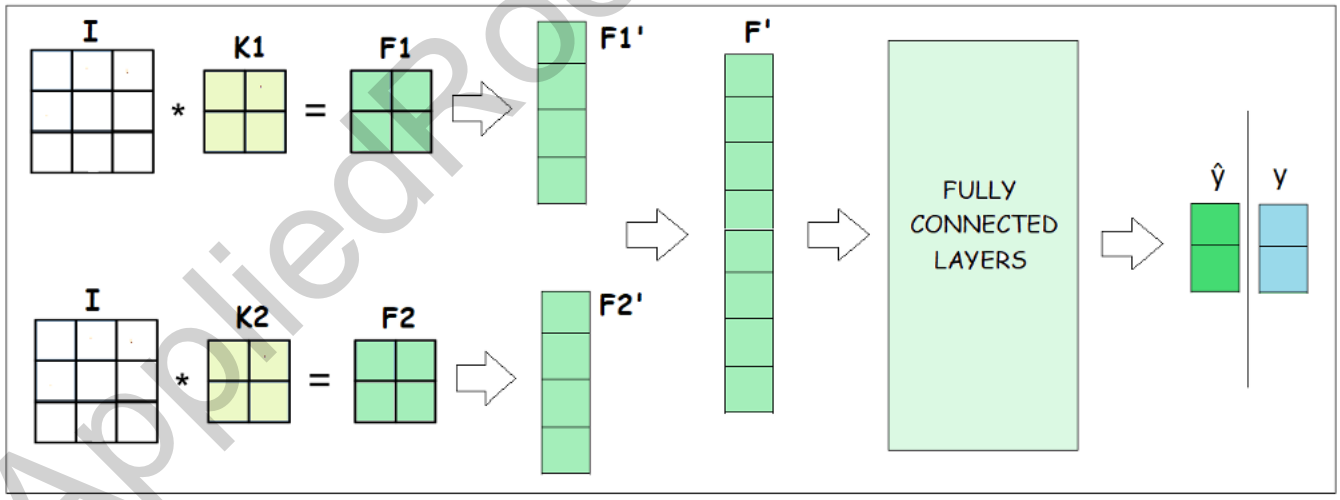
LOCAL RECEPTIVE FIELD(LOCAL PATTERNS VERSUS GLOBAL PATTERNS):

The kernel size defines the size of the window that is moved over the input image matrix. It is within this "window" that we seek to detect patterns. In other words, this window represents the "**local receptive field**" of the CNN. Thus we could say that every value in the feature map is representative of a particular position of this local receptive field on top of the input image matrix.

The concept of Local receptive fields allows one to detect **local patterns** since only a part of the image is accessed during each step of the convolution. Thus the weights of the kernel are shared across the entire image. This is unlike standard neural networks, where each neuron in the layers is connected to all of the input values. This kind of architecture considers information from the entire image to tune its weights and thus detects **global patterns**.

DETECTING MULTIPLE FEATURES:

Consider an image matrix I on which we perform convolutions using multiple same sized kernels, where each kernel's weights have been randomly initialized, while at the same time making sure that they are different for each kernel. This makes each kernel detect a different pattern. Each kernel produces a feature map that is indicative of the unique pattern it has learnt. Each of these feature maps are then flattened and concatenated to form a single vector, which is then fed into a fully connected neural network as shown in the image below.



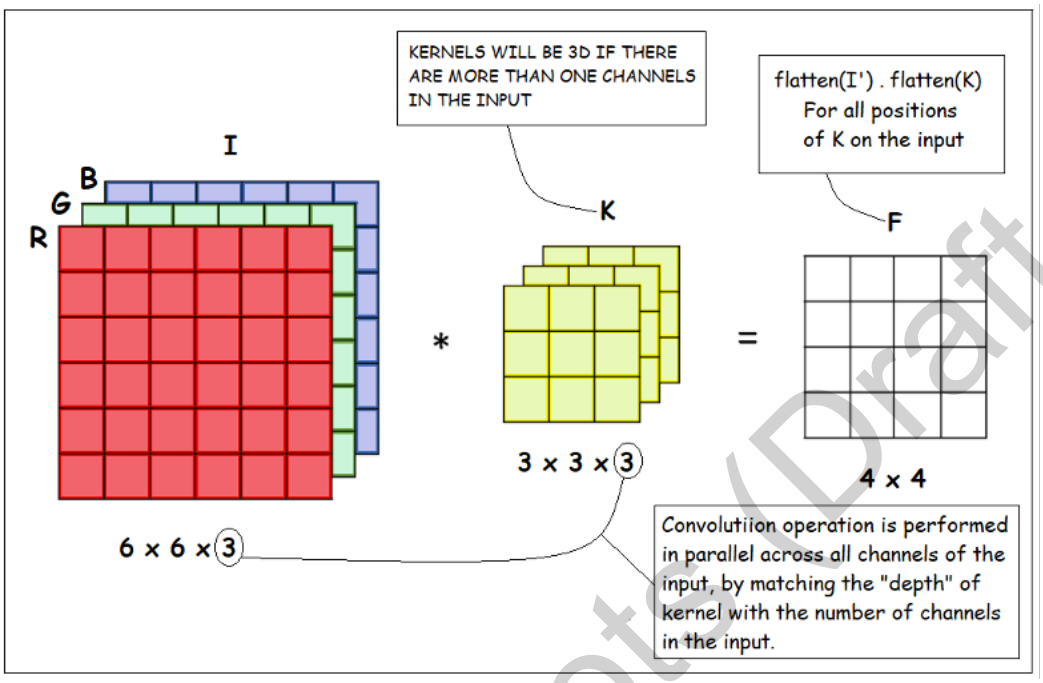
Thus we can have multiple kernels that learn to detect different local patterns that are common to all the images in the dataset.

MULTI CHANNEL INPUTS & THEIR CORRESPONDING KERNELS:

Images can be of two types - colored and grayscale. Colored images are numerically represented by three 2D matrices (3D tensor/array), where the three layers represent various intensities of red, green and blue colors (RGB) in the image. Each of these RGB layers is called

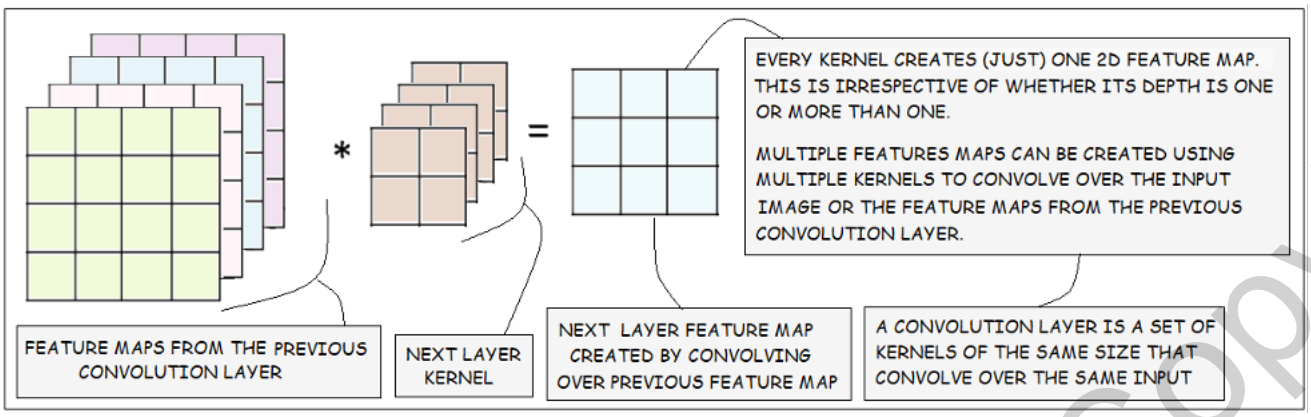
a “channel”. Thus coloured images have three channels, whereas grayscale images have only one channel.

Convolution over a grayscale image is the same as the convolution operations discussed till now, where the input is a 2D matrix. But for convolution over colored images, the kernel used is 3 layered (one for each channel in the input) as shown in the image below.



As described in the image above, the kernel simultaneously moves/convolves over each of the channels of the input image, because it too has the same number of channels as the input. At each position of the kernel the dot product as shown in the image above is computed and used to populate a 2D feature map **F** (ie: the dot product between the flattened 3D kernel **K** and the flattened 3D array **I'** that represents the part of the input array that corresponds to the current position of the kernel).

The convolution operation described above can be generalized to inputs that have more than three channels. This occurs when we perform the convolution operation over the feature map outputted by a previous convolution operation that used multiple kernels. Consider the case where we use four 3x3x3 sized kernels instead of just one as shown in the above image. Then the output of such a convolution will be a feature map having four channels. The image below describes the convolution operation performed on the feature map that is the outcome of the convolution just described.



INTRODUCING NON LINEARITY (RELU):

As discussed in previous chapters the ReLU activation function is used to introduce non linearity in the model. Without non linear activation, the kernel would be just performing a linear transformation. By passing each of the dot products that happen for all positions of K,

through the ReLU function, we in essence make it possible for us to learn complex nonlinear mappings between the initial inputs (images) and the desired outputs (labels).

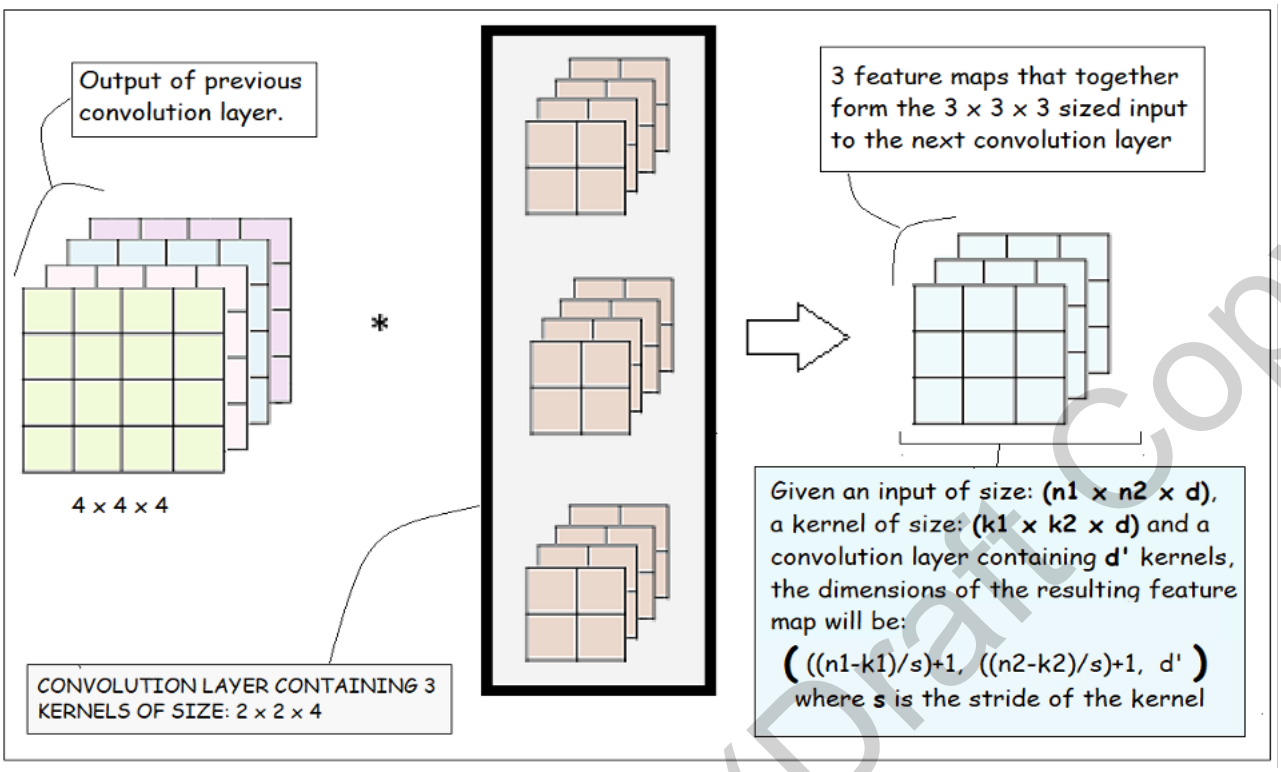
In other words, each Feature Map can be the input to another set of kernels and so on, until a satisfactory depth. The final feature map resulting due to such a series of convolutional operations is then flattened and fed through a multilayered conventional neural network, which outputs a prediction. Introducing non linearity makes it possible for us to detect and learn even quite complex non linear mapping that can occur between the images and the labels.

Thus the convolution operation used is:

$$ReLU(\text{flatten}(I') \cdot \text{flatten}(K)) \quad \text{For every position of K on I}$$

CONVOLUTION LAYER :

A convolution layer is a set of kernels Of the same size that convolve over the same input as shown in the image below.

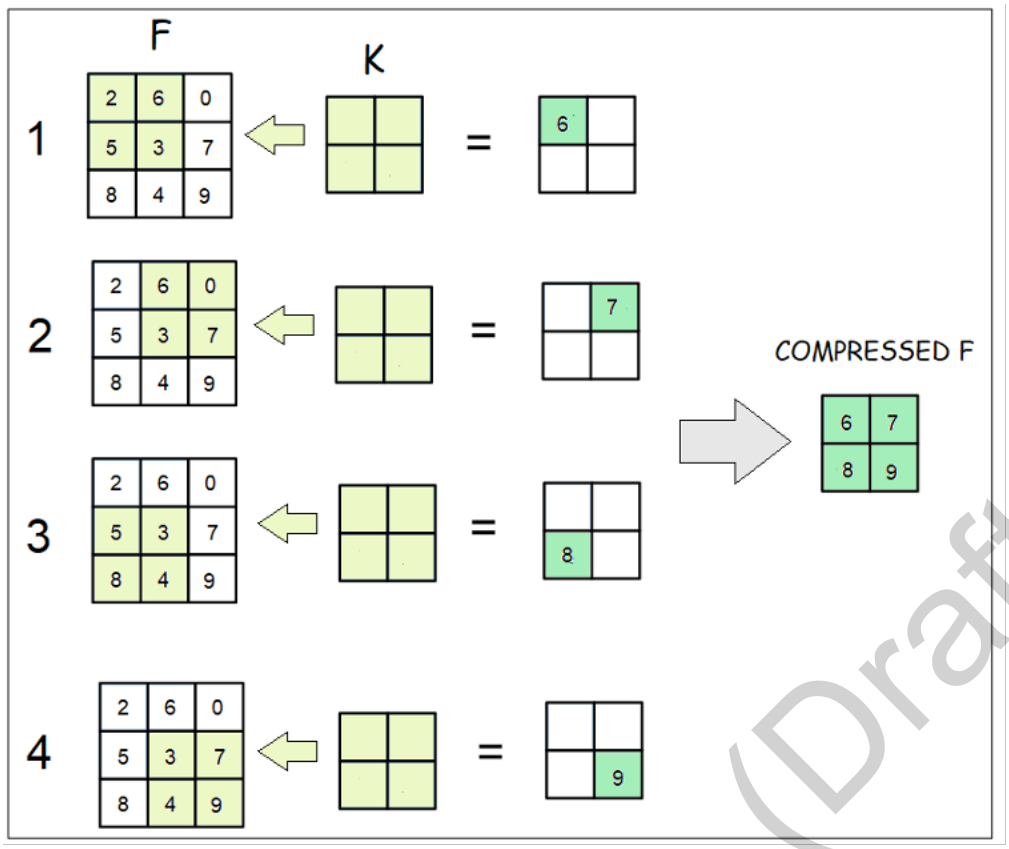


Due to the fact each kernel in the layer will be initialized differently, they will each learn different patterns (of the same size) from the image/feature map it is exposed to. The structure of a convolution layer is such that:

- The depth of each kernel is the same as the depth of its inputs.
- Each kernel independently/parallely performs the nonlinear convolution operation discussed just previously, on its inputs (ie: images or feature maps).
- Every layer produces a 3D feature map, whose dimensionality will be a function of the input size, the kernel size and the stride - as shown in the image above.
- Multiple layers could be used in sequence.

POOLING LAYERS:

The pooling operation is similar to the convolution operation, except that the kernel in this instance does not have any weights. Instead, for every position of the pooling "kernel" on the feature map outputted by the convolution layer, it computes some kind of aggregate of the values it covers. Usually the aggregate value pooled is the maximum value as shown in the image below. Other pooling operation such as geometric average pooling or harmonic average pooling can also be performed, but these are rarely used.



Pooling layers are used immediately after every convolutional layer. These layers simplify the information and reduce the scale of feature maps. In other words, pooling layers create a compressed and a more generalized feature map from the original feature map produced by the convolutional layers, thus reducing the overfitting issues in the CNNs.

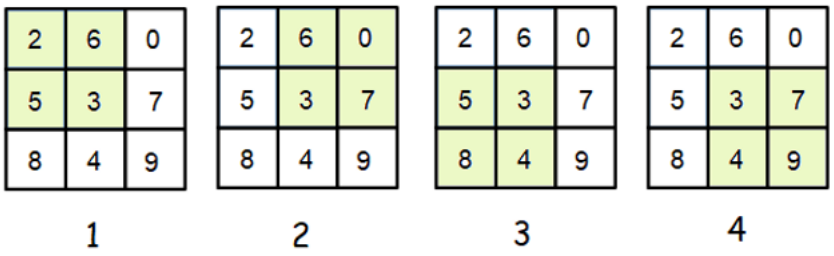
Since the pooling layer basically collects aggregated information from various zones within the feature map, it imparts the property of "position invariance" to the CNN. In other words, the model will be able to recognize an object as an object, even when its appearance varies in some way (ie: Variations in the relative positions of the camera and the object in the image, for example the model will be recognise a image of car irrespective of differences in the viewing angle or position/orientation of the car in the image).

PADDING:

The convolution operation/layer that has been discussed till now has some inherent drawbacks:

1. The image/feature map shrinks every time a convolution operation is performed. Consider an 8 x 8 size image and 3 x 3 kernel, the output resulting after convolution operation would be of size 6 x 6. This attribute places a constraint on the number of times such an operation could be performed before the image reduces to nothing thereby restricting us from building deeper networks

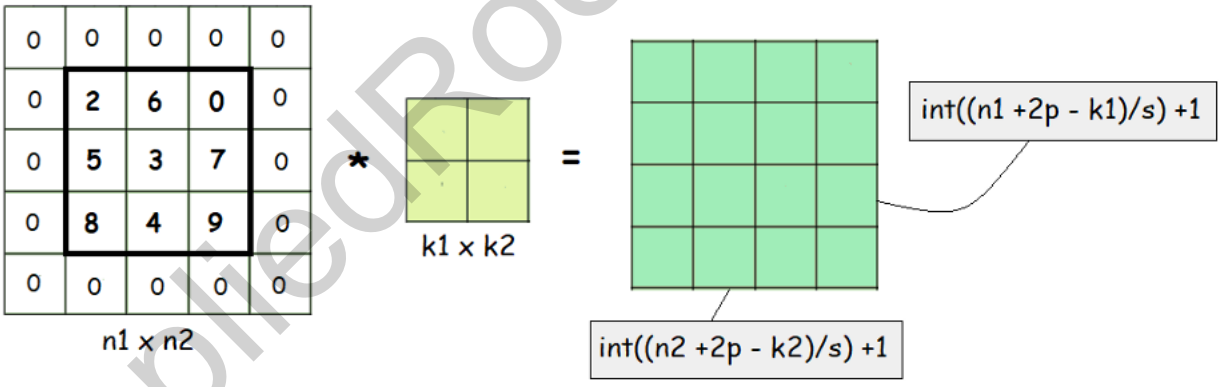
2. The pixels on the corners and the edges of the images/feature maps are “used” much less than those in the middle. Consider the convolution shown below.



The values at the corners (ie: 2, 0, 8 & 9) are exposed to the kernel just once, whereas the value 3 at the center gets exposed the most (it is in the kernel for all four positions of the kernel). The result being that the information on the borders of images/feature maps are not preserved as well as the information in the middle.

To overcome the problems mentioned, we use padding. Padding basically involves creating a set of rows and columns around the original image/feature map as shown in the image below and then populating them with zeroes.

Padding results in a bigger sized feature map and it also results in the usage of the edge values in the input more effectively.



KERAS CONVOLUTION AND POOLING LAYERS:

For most of our image convolution purposes we will be using the Conv2D class defined within Keras. Shown below is the Conv2D layer with the most usually used settings of some of the most relevant parameters & keyword arguments. The “filter” parameter is used to define the number of kernels one chooses to have. The kernel size defined in the form of a tuple (ex: (2, 2)), lets us set the window size of the kernel.

```
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides = (1, 1),
    padding = 'same',
    activation = 'RelU',
    use_bias = True,
    kernel_initializer = 'glorot_uniform')
```

Ths MaxPooling2D class defined within keras helps us define pooling layers. Shown in the image below is the MaxPooling2D layer with the most usually used settings for its parameters & keyword arguments

```
tf.keras.layers.MaxPooling2D(
    pool_size = (2, 2),
    strides = (1, 1),
    padding = 'same')
```

For the "padding" keyword argument in both the layers shown above, keras provides two alternatives: "valid" and "same". The former means no padding will be used and the later means a padding configuration will be used such that the size of the feature map outputted by the convolution/pooling layer will be the same as the input image/featuremap size and the values contained within the padding rows/columns will be zeroes.

IMAGE CLASSIFICATION USING CNNS (MNIST):

We will train a CNN classifier on the MNIST dataset for demonstration purposes. As discussed before in the chapters on tSNE dimensional reduction, the **MNIST database** (Modified National Institute of Standards and Technology database) is a database of 70,000 images of handwritten digits (1 to 9). Each image is 28 x 28 pixel

THe MNIST dataset can be accessed as shown below.

```
(X_tr, y_tr), (X_ts, y_ts) = keras.datasets.mnist.load_data()

sample = X_tr[0]
classes = np.unique(y_tr)

sample.min(), sample.max(), classes
```

```
(0, 255, array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8))
```

As can be seen from the output above, the values of X lie in the [0, 255] range. We normalize the values to be be in the [0, 1] as shown below:

```
X_tr = X_tr.astype("float32") / 255
X_ts = X_ts.astype("float32") / 255
```

Since the input to the CNN will should of the shape : (image size, number of channels), we reshape the train & test inputs as shown below:

```
X_tr.shape, y_tr.shape, X_ts.shape, y_ts.shape
```

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

```
X_tr = X_tr.reshape(*X_tr.shape, 1)
X_ts = X_ts.reshape(*X_ts.shape, 1)

X_tr.shape, X_ts.shape
```

```
((60000, 28, 28, 1), (10000, 28, 28, 1))
```

We then define our CNN model using the convolution and pooling layers as shown below.

```
num_classes = len(classes)
input_shape = X_tr[0].shape

num_classes, input_shape
```

```
(10, (28, 28))
```

```
import tensorflow as tf
from tensorflow import keras

model_1 = keras.Sequential(name = 'CNN_1')

model_1.add(keras.Input(shape=input_shape))
model_1.add(keras.layers.Conv2D(32, kernel_size=(3, 3), padding = 'same', activation="relu"))
model_1.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model_1.add(keras.layers.Conv2D(64, kernel_size=(3, 3), padding = 'same', activation="relu"))
model_1.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
model_1.add(keras.layers.Flatten())
model_1.add(keras.layers.Dropout(0.2))
model_1.add(keras.layers.Dense(100, activation="relu"))
model_1.add(keras.layers.Dropout(0.2))
model_1.add(keras.layers.Dense(num_classes, activation="softmax"))

model_1.summary()
```

```
Model: "CNN_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dropout (Dropout)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 100)               313700
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1010
=================================================================
Total params: 333,526
Trainable params: 333,526
Non-trainable params: 0
```

We then find the optimal weights to balance the classes as shown below:

```
from sklearn.utils import class_weight

classes = np.unique(y_tr)
balanced_class_weights = class_weight.compute_class_weight('balanced', classes, y_tr)
balanced_class_weights = {k:v for k, v in enumerate(balanced_class_weights)}
balanced_class_weights
```

```
{0: 1.0130001688333614,
 1: 0.8899436369029962,
 2: 1.0070493454179255,
 3: 0.978633175664655,
 4: 1.0270455323519343,
 5: 1.1068068622025458,
 6: 1.013856032443393,
 7: 0.9577015163607342,
 8: 1.025465732353444,
 9: 1.0085728693898135}
```

We then train and evaluate the CNN model using the function "**fn_CNN_multi_clf**" which is similar to the training function used in the previous chapters on neural networks, as shown below.

```
model = model_1
optimizer=tf.keras.optimizers.Adamax()

model_save_path =  data_path + 'CNN_MODELS/model_1/'
batch_size, epochs = 128, 20
class_weight = balanced_class_weights

df_perform_cnn = fn_CNN_multi_clf(model, optimizer, X_tr, y_tr, num_classes,
                                  model_save_path = model_save_path, batch_size = batch_size,
                                  epochs = epochs, class_weight = class_weight)

df_perform_cnn.describe()
```

100% 20/20

	loss	acc	val_loss	val_acc	diff_loss	diff_acc
count	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000
mean	0.063123	0.980764	0.046145	0.986538	0.024729	0.007420
std	0.083250	0.024266	0.021876	0.006418	0.059580	0.017500
min	0.019234	0.884021	0.032007	0.964000	0.000169	0.000313
25%	0.026465	0.981859	0.034469	0.985354	0.005589	0.001474
50%	0.037200	0.988552	0.038195	0.988958	0.009167	0.002573
75%	0.058423	0.991542	0.047658	0.990104	0.013452	0.003729
max	0.395411	0.993521	0.122510	0.991000	0.272901	0.079979

We then filter for the best models as shown below.

```
dff = df_perform_cnn

df1 = dff[dff.diff_acc < 0.006]
df2 = df1[df1.acc > 0.993]
df_filtered = df2[df2.loss < 0.04]

df_filtered
```

	loss	acc	val_loss	val_acc	diff_loss	diff_acc
CNN_1_epoch_18	0.020932	0.993187	0.032038	0.990500	0.011106	0.002688
CNN_1_epoch_19	0.019234	0.993521	0.032132	0.990417	0.012899	0.003104

We choose model_19 and check for generalization to the test set as shown below.

```
X_  = X_tr
y_ = y_tr
model_ = keras.models.load_model(model_save_path + 'CNN_1_epoch_19.h5')

fn_test_model_multi_clf(X_, y_, model_, NN = True)
```

```
==================
ACCURACY: 99.622
LOGLOSS: 0.014
==================
```

	precision	recall
0	99.781	99.831
1	99.718	99.778
2	99.598	99.832
3	99.690	99.641
4	99.573	99.692
5	99.375	99.650
6	99.612	99.747
7	99.522	99.601
8	99.828	99.163
9	99.495	99.260
MICRO	99.622	99.622
MACRO	99.619	99.619

```
X_, y_ = X_ts, y_ts

fn_test_model_multi_clf(X_, y_, model_, NN = True)
```

```
==================
ACCURACY: 99.12
LOGLOSS: 0.025
==================
```

	precision	recall
0	98.986	99.592
1	99.298	99.648
2	98.844	99.419
3	99.112	99.406
4	99.492	99.695
5	98.328	98.879

6	99.579	98.747
7	99.120	98.638
8	99.481	98.460
9	98.907	98.612
MICRO	99.120	99.120
MACRO	99.115	99.110

As can be seen from the results shown above, the model achieves an accuracy of 99% and it generalizes well to the testset.

FEATURE COMPOSITIONALITY IN CNNS:

If one were to visualize the feature maps created by the convolution/pooling layers in deep CNNs, it would be observed that the earlier feature maps consist of "visual primitives" like lines, curves, edges and corners. As we proceed deeper, the feature maps tend to get more complex and usually consist of elementary shapes. Finally the feature maps corresponding to the deepest layers show more complex patterns, depicting parts of the original image that it was trained on (ex: Nose, ears, eyes when trained on images of faces). This hints at a progressive compositionality in CNNS, where it starts with recognizing basic **local** visual primitives in the earlier layers. Each layer then progressively uses the patterns of the previous feature map such that at the deeper layers more complex **global** patterns are being detected.

03. ADVANCED PRE-TRAINED IMAGE CLASSIFIERS

ADVANCED PRE-TRAINED IMAGE CLASSIFIERS

IMAGE NET:

Image net is a large visual database designed for use in visual object recognition. It consists of more than 14 million labelled images that contain around 22,000 separate object categories. The database is associated with the "ImageNet Large Scale Visual Recognition Challenge", or **ILSVRC** for short. The aim of this challenge is to design/train machine learning/deep learning models that can classify an input image into 1,000 separate object categories as accurately as possible.

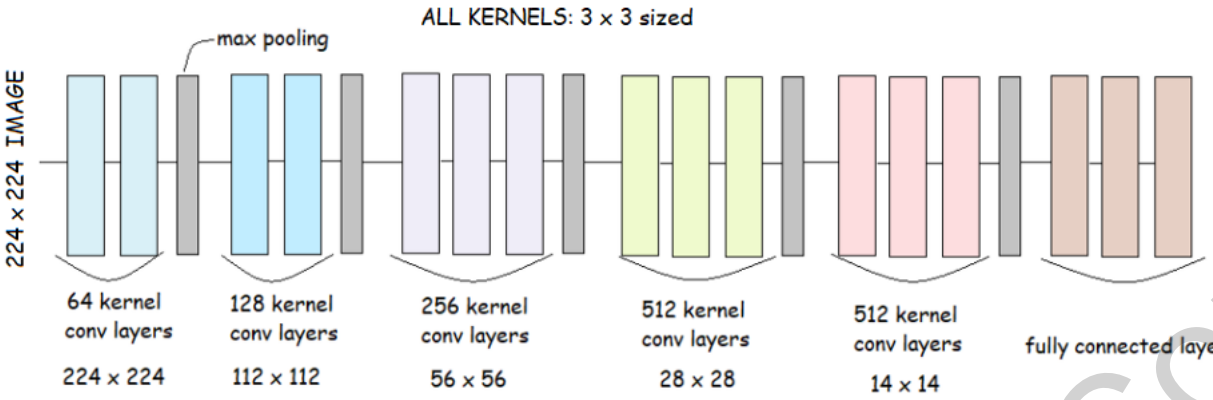
These 1,000 image categories represent object classes that we encounter in our day-to-day lives, such as various household objects, vehicle types, species of dogs, cats and much more.

The ImageNet challenge has turned out to be the de facto benchmark for computer vision classification algorithms and it has resulted in some state of the art CNNs architectures and pretrained models like VGG, Resnet and Inception.

VGG16:

VGG16 is one of the simplest CNN architectures used in the Imagenet competition. It's Key characteristics are:

- This network contains a total of 16 layers as shown in the image below.
- The number of kernels in the convolution layers follow an increasing pattern.
- The kernels used in the convolution layers are always 3 x 3
- Padding for all the convolution layers is "same".
- Max pooling layers are applied between each group of convolution layers (Colored coded below for different sizes). The kernels used for pooling are always 2 x 2 with stride 2, thus the size of the feature maps is divided by 2 after each pooling layer.
- The dense layers comprises 4096, 4096, and 1000 (for 1000 classes) nodes each.



The VGG models achieved state of art performance in the 2014 imagenet challenge when 16 and 19 layer networks were considered very deep. We now have resnet models that have successfully trained at depths of more than a 100 layers on imagenet.

RESNET:

In the previous discussion we have seen that deep convolutional networks are capable of learning a hierarchy of features starting from basic patterns in the earlier layers to more complex patterns (that are composition of previous patterns) in the deeper layers. In general the depth of the neural network plays a crucial role in the effectiveness of the neural network.

In fact, the key takeaway from the previous chapters of ANNs is that one should strive to build as deep a neural network as possible (over parameterize) and use regularization techniques like drop out and batch norm to overcome the problem of overfitting that arises with increased depth.

But creating better neural networks is not as simple as stacking more layers, due to the problems of vanishing and exploding gradients that become more evident as the depth of the network is increased. Using the ReLU activation function, batch normalization and proper weight initialization techniques does decrease this problem to an extent, but is not enough as network depths get deeper. It was noticed that there is a limit to the "effectiveness of depth" and that after a certain threshold, increasing the depth of a network led to lower performances than the shallower models.

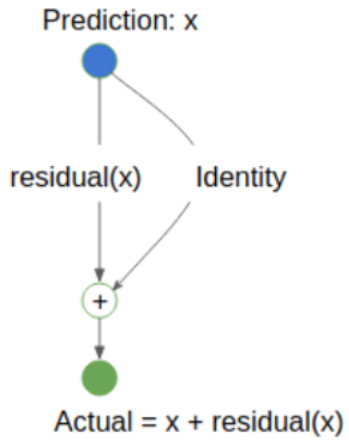
The Resnet CNN architecture was designed to overcome this problem.

SKIP CONNECTIONS:

The creators of the resnet architecture came up with the idea of skip connections with the hypothesis that the deeper layers should at least be able to learn something as equal as the shallower layers. Here the concept of identity function and "residual layers" are used such that the deeper network will still be equivalent to the shallower network despite the added depth

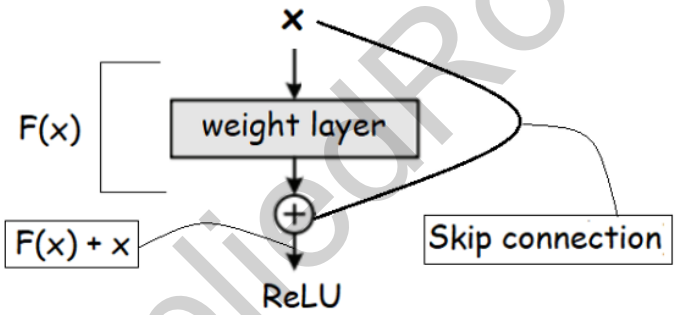
and hence should not produce lower performance than its shallower counterpart.

Consider the mathematical definition of a residual. It can be expressed with the help of the image shown below. A residual is the error in a result.



In the diagram, **x** is the prediction and we want it to be equal to the Actual. However, if it is off by some margin, the residual function **residual()** will kick in and produce the residual of the operation so as to correct the prediction and make it match the actual. If **x** is the same as Actual, residual(x) will be 0. The Identity function simply copies **x** as shown above.

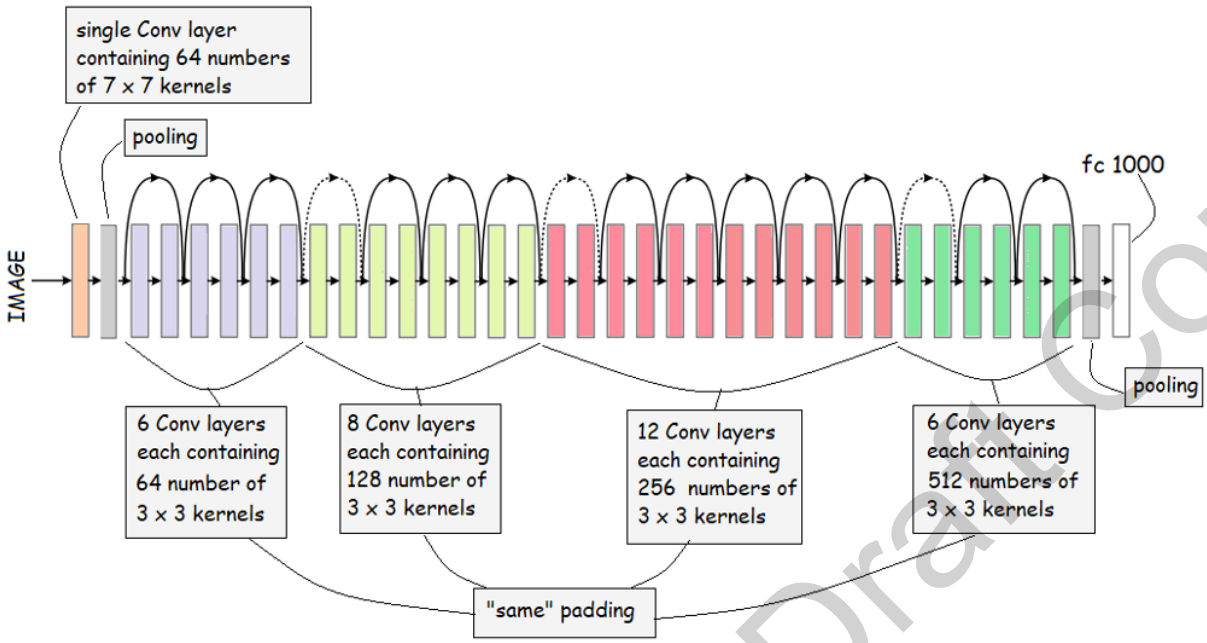
The above logic is incorporated into deep neural networks as shown in the image below.



The role of skip connection is to perform the identity function. In other words, the output of a previous layer is added with the output of some deeper layer as shown in the image above (The addition is performed before the activation function of this deeper layer) . Thus the neural layer(s) that lie between the skip connection perform the job of the residual function. The weights learnt by these layers will reflect the residual and will amount to nothing if the added deeper layers do not improve the performance of the model.

To enable these skip connections (addition operation), one needs to ensure the same dimensions of convolutions throughout the network, that's why resnets have the same 3 by 3 convolutions with "same" padding throughout.

Shown in the image below is the 32 layered Resnet initially designed by the creators of Resnets.



The authors of the paper were able to create the deep neural network architecture with 152 layers. The variants of Resnets such as resnet34, resnet50, resnet101 have produced the solutions with very high accuracy in Imagenet competitions.

INCEPTION NETWORK:

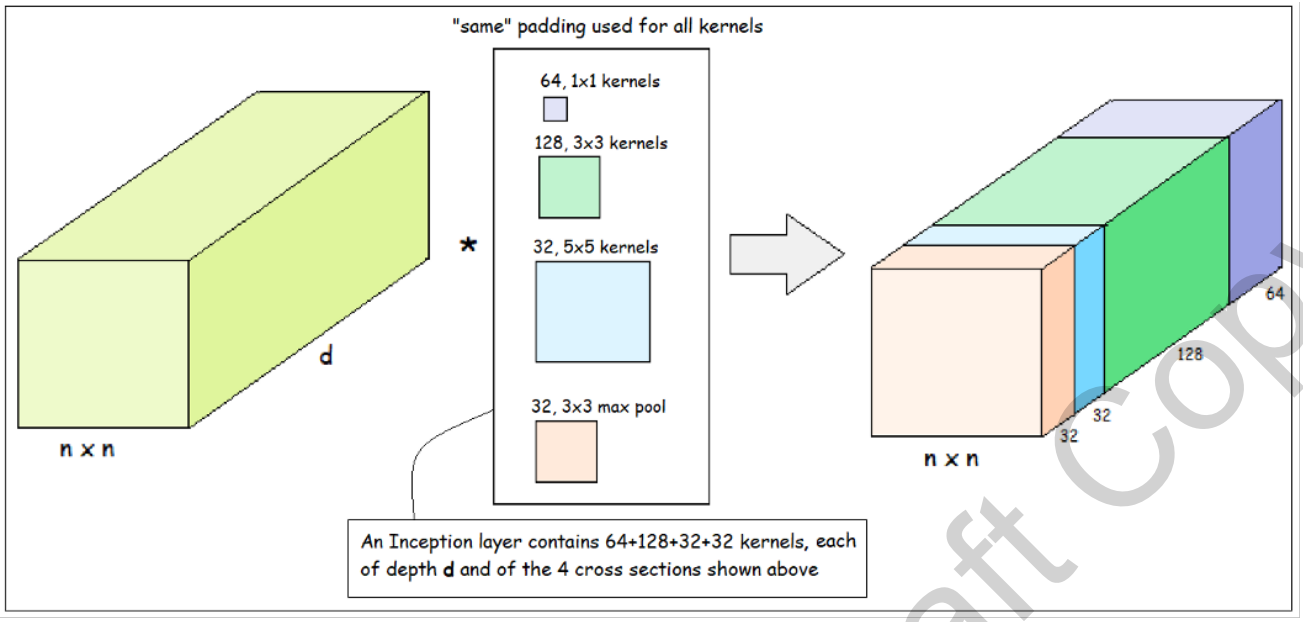
In all the CNNs until now, the sizes of the kernel used in their layers were homogeneous (ie: A layer will contain only same size kernels). But what if we simultaneously wanted to try out a set of kernels of different sizes? In other words it is not easy to know what the optimal size of the local receptive field of a CNN is. Different images could have different sized patterns within them and so it would be a good strategy to try out a set of most useful kernel sizes simultaneously. It is this requirement that the inception network aims to satisfy.

Consider the image shown below. The convolution layer contains 3 different sized kernels and a max pooling layer. All of these kernels convolve over the same input and learn different sized patterns from it.

A combination of the following set of kernels is generally used in inception networks.

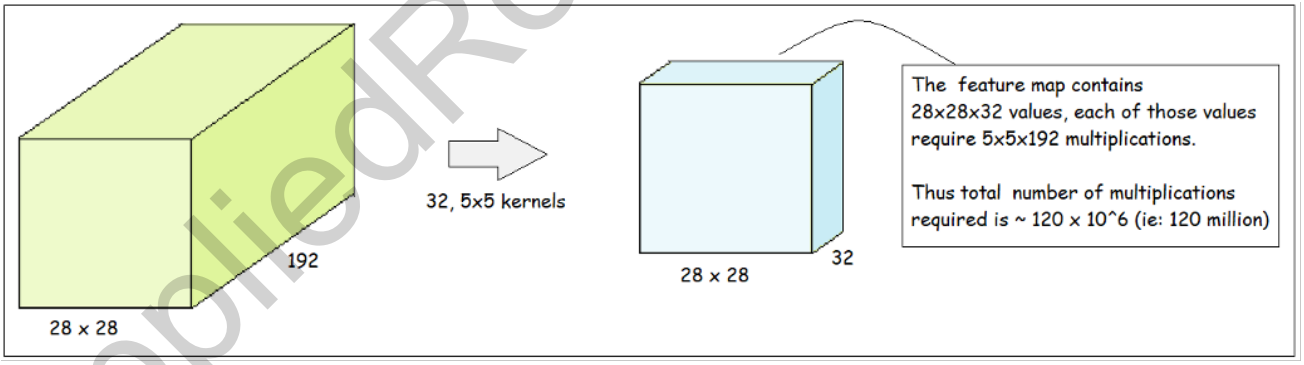
1. 64, 1x1 kernels
2. 128, 3x3 kernels
3. 32, 5x5 kernels
4. 32, 3x3 max pooling kernels.

All of the above kernels use “same” padding over the input. This leads to a feature map of the same cross section as the input and having 256 channels.

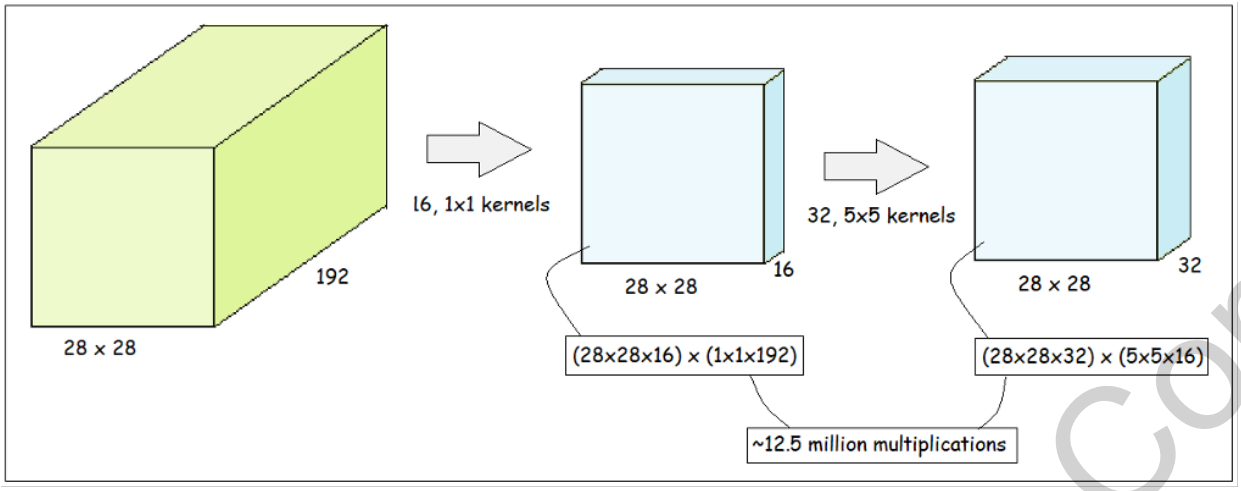


THE 1 X 1 CONVOLUTION FILTER (REDUCING COMPUTATIONAL COST):

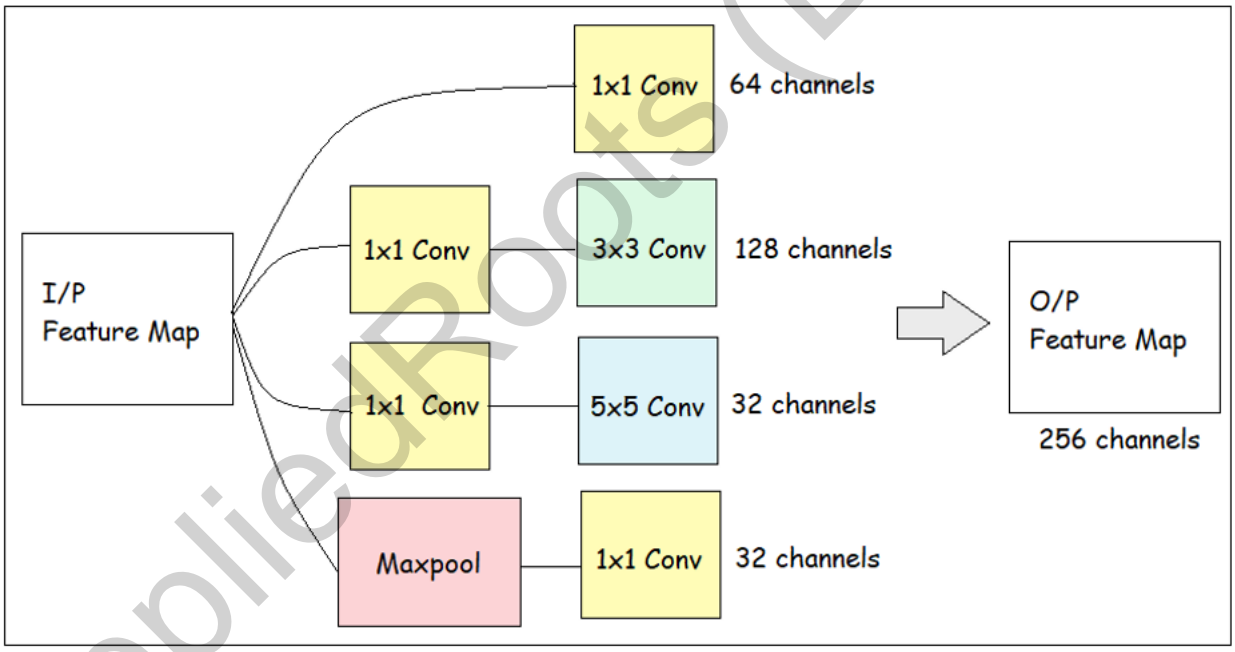
Consider the amount of computation required while using the 5x5 kernel shown in the image above. This would take 120 million multiplication operations (as explained in the image below). This is quite large by itself, now if we consider the amount of computation that would be performed for the entire layer of kernels in the image shown above, it would be even larger. Thus Inception network would be computationally expensive. This is where 1x1 convolution kernels come in handy.



Consider the image shown below. By using a 1x1 sized kernel in the way depicted, we can reduce the amount of computations by an order of magnitude and still produce the same sized output. Thus in the inception networks 1x1 kernels are advantageously used to bring the computational cost down.



Shown in the image below is the outline of the inception network with 1x1 convolution kernels incorporated to reduce computation cost. This would in essence be a single inception module/ layer. Many such inception modules can be used one after another to produce powerful CNN models.



Note that the 1x1 conv kernel used after the max pooling kernel is used to create a feature map of 32 channels from the single channel feature map that it outputs.

The GoogleNet CNN which was the winning model of 2014 image net challenge used a total of 22 such inception modules/layers to perform its task

TRANSFER LEARNING (USING PRE TRAINED MODELS):

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. For example, a multi class classification

model that was trained on a huge dataset of animal photos, can be further be used by others on slightly different but similar tasks such as classifying cats and dogs, using a much smaller dataset. In other words, a model trained on one domain is adapted to perform tasks in another different but related domain.

Transfer Learning makes it possible for the general deep learning practitioner to use state of the art models created by big companies (like the VGG, Resnet or inceptions models) and use them for their own tasks with a little amount of model fine tuning.

We know from our previous discussion on "compositionality" in CNN layers that the features/ patterns learnt are more generic in early layers and more "original dataset specific" in later layers. Thus if we can remove the last few layers of the original state of the art CNN model and replace them with a set of untrained layers and if we trained only the new layers of this modified model on a new task, while keeping the rest of the pretrained weights in the other layers intact, it is quite possible to get a high performance model for our new task.

The procedure for Transfer Learning can be boiled down to the following steps:

1. Download the preferred state of the art pretrained model (VGG, Resnet, Inception net etc).
2. Remove the last layer of the model (ie: the predictive layer).
3. Create a new CNN model using the remaining layers.
4. Freeze the weights of this new CNN model (make them untrainable). Therefore they still retain what was learnt when they were trained on the imagenet data to produce cutting edge performance.
5. Add a set of new layers to this new model
6. Train the new model on the dataset specific to our task (only the newly added layers will be trained).

CLASSIFICATION OF DOGS & CATS IMAGES (TRANSFER LEARNING USING VGG16):

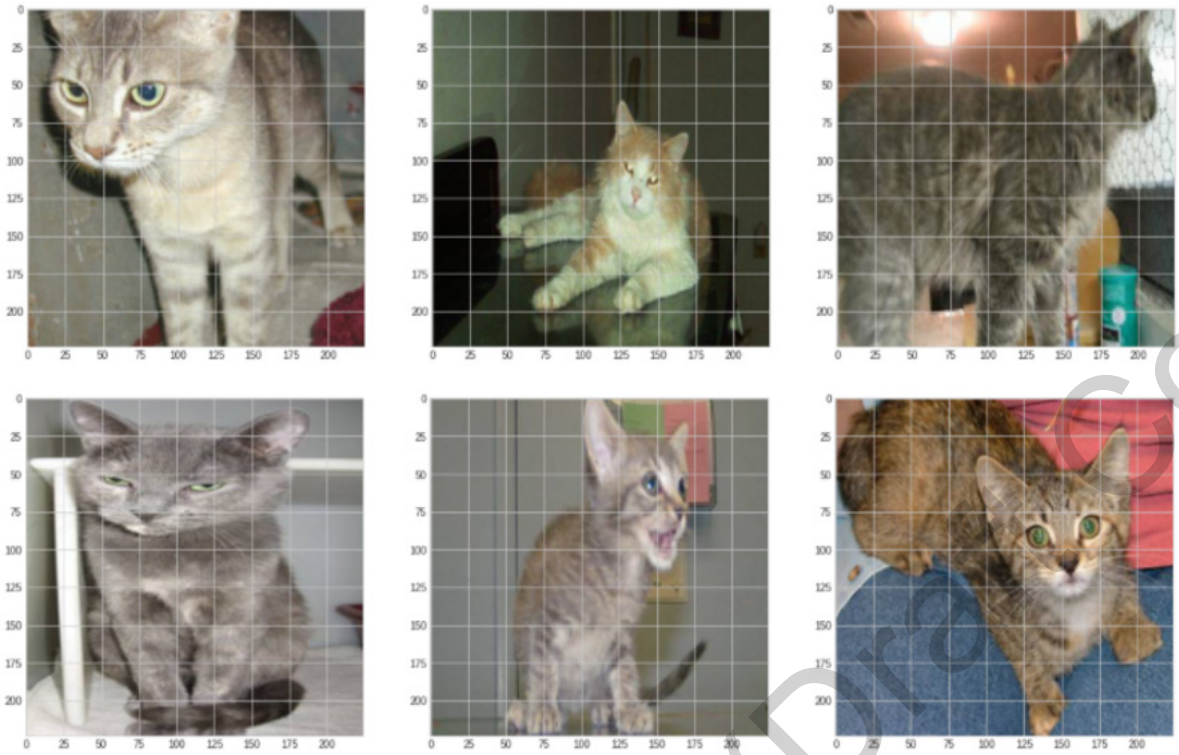
The Keras core library contains some of the high performance Convolutional Neural Networks from the ImageNet challenge over the past years. We shall use the VGG16 pre trained model for our task of classifying images of cats and dogs. For this purpose we will be using a small subset (1300 images) of the Cats & Dogs dataset presented at the Kaggle machine learning competition, which contains 25000 labelled images of cats and dogs.

INSPECTING IMAGES OF CATS & DOGS:

```
data_path = 'gdrive/My Drive/3A_PGD/CODE_PGD/CNNs/'

tr_path = 'dogs_cats/train/'
ts_path = 'dogs_cats/test/'
val_path = 'dogs_cats/valid/'

dog_tr_path = 'dogs_cats/train/dog/'
cat_tr_path = 'dogs_cats/train/cat/'
```

```
dog_tr_imgs = fn_files_in_folder(data_path + dog_tr_path)

fn_show_images(dog_tr_imgs)
```

```
cat_tr_imgs = fn_files_in_folder(data_path + cat_tr_path)

fn_show_images(cat_tr_imgs)
```



KERAS DATA LOADERS:

Keras offers some well designed data loaders that help generate the train data from the machine's data storage one batch at a time instead of loading the entire dataset into the RAM. For images, there is the ImageDataGenerator class that generates batches of image data in numerical format while performing the specified preprocessing tasks (for ex: resizing, scaling and many other processes/augmentations) in real time as the image data is being fed to the model. The ImageDataGenerator class and some of its more relevant default specifications are shown below.

```
tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    rotation_range=0,
    brightness_range=None,
    shear_range=0.0,
    zoom_range=0.0,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    preprocessing_function=None,
    dtype=None)
```

The various specifications of the class shown above are basically for either data normalization or "data augmentation" which basically is a process of recreating variations of the same image by rotating, flipping, changing brightness and other such techniques.

The "**preprocessing_function**" specification allows us to use predefined (by us or by keras) functions that perform all the necessary image preprocessing.

Instances of the ImageDataGenerator class have associated with them various methods for specifying where the data should be generated from. The **flow_from_direcctory** method allows one to generate data from files stored in some directory in the machine's permanent memory. Its default setting is shown below.

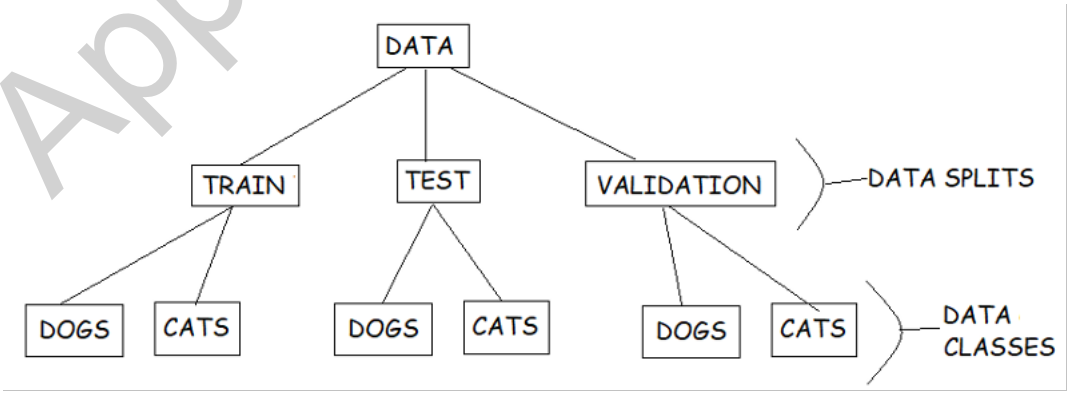
```
.flow_from_directory(
    directory,
    target_size=(256, 256),
    color_mode="rgb",
    classes=None,
    class_mode="categorical",
    batch_size=32,
    shuffle=True)
```

The **target_size** specification lets us specify the dimensions to which all images found in the directory will be resized to. The **flow_from_direcctory** method can detect PNG, JPG, BMP, PPM and TIF images stored in the directory specified.

The **color_mode** specification lets us specify the input channels ('rgb' or 'grayscale').

The **classes** specification lets us specify the name of the classes in the dataset.

The **class_mode** specification lets us specify the loss function that will be used during training, so that keras can "shape" the labels values in the format required for these losses (ie: as one hot encoded vectors or as simple 0, 1 labels)



The **flow_from_directory** method expects the data to be stored in folders in a hierarchy as shown above. The names of the folders on the lowest level (cats & dogs) should match with what is provided as arguments for the "**classes**" specification within the method.

We thus create image loaders of each data split in the dataset as shown below:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

kwarg_img_gen = dict(preprocessing_function=tf.keras.applications.vgg16.preprocess_input)
kwarg_flow_from_directory = dict(target_size=(224,224), classes=['cat', 'dog'], batch_size=10)

tr_datagen = ImageDataGenerator(**kwarg_img_gen)
ts_datagen = ImageDataGenerator(**kwarg_img_gen)
val_datagen = ImageDataGenerator(**kwarg_img_gen)

tr_datagen = tr_datagen.flow_from_directory(directory = data_path+tr_path,
                                            **kwarg_flow_from_directory)
val_datagen = val_datagen.flow_from_directory(directory = data_path+val_path,
                                              **kwarg_flow_from_directory)
ts_datagen = ts_datagen.flow_from_directory(directory = data_path+ts_path,
                                            shuffle=False,
                                            **kwarg_flow_from_directory)

```

```
Found 1000 images belonging to 2 classes.
Found 200 images belonging to 2 classes.
Found 101 images belonging to 2 classes.
```

As can be seen from the image above, we have used a predefined preprocessing function within the ImageDataGenerator (**tf.keras.applications.vgg16.preprocess_input**). This predefined function preprocesses the input such that it is ready to be fed to the VGG16 classifier.

Note that, within the flow_from_directory method we specify:

A. The size the image should have before being fed to the model.

B. The classes (name of the sub folders within the "train", "valid" and "test" folders - dogs and cats).

C. The batch size.

The output of the code above tells us that the ImageDataGenerator we created was able to detect the data and is now ready to generate the train, validation & test data when required.

We then download the vgg16 model as shown below:

```
pretrained_model = tf.keras.applications.vgg16.VGG16()
```

```
Downloading data
553467904/553467096 [==============================] - 3s 0us/step
```

We create a new sequential model using all the layers (except the last) of the pretrained VGG16 model as shown below:

```
model = Sequential()
for layer in pretrained_model.layers[:-1]:
    model.add(layer)
```

We then "freeze" the weights within these layers as shown below:

```
for layer in model.layers:
    layer.trainable = False

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
=================================================================
Total params: 134,260,544
Trainable params: 0
Non-trainable params: 134,260,544
_________________________________________________________________
```

We can see from the above model summary that all its parameters are frozen (non trainable) and thus retain the configuration of the original high performing model that was trained on Imagenet.

We then add a set of new fully connected layers to the above model as shown below.

```
model.add(Dense(units=100, activation='relu'))
model.add(Dense(units=50, activation='relu'))
model.add(Dense(units=2, activation='softmax'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
dense (Dense)                (None, 100)               409700
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 102
=================================================================
Total params: 134,675,396
Trainable params: 414,852
Non-trainable params: 134,260,544
_________________________________________________________________
```

As can be seen from the above summary the model now has 3 new dense layers attached to it with 4,14,852 trainable parameters.

We then compile the above model as shown below:

```
optimizer=Adam(learning_rate=1e-4)
loss='categorical_crossentropy'
metrics=['accuracy']

model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
```

We train and evaluate the model as shown below:

```
model.fit(x=tr_datagen,
          steps_per_epoch=len(tr_datagen),
          validation_data=val_datagen,
          validation_steps=len(val_datagen),
          epochs=5,
          verbose=2)
```

```
Epoch 1/5
100/100 - 9s - loss: 0.1356 - accuracy: 0.9410 - val_loss: 0.0525 - val_accuracy: 0.9750
Epoch 2/5
100/100 - 6s - loss: 0.0170 - accuracy: 0.9960 - val_loss: 0.0584 - val_accuracy: 0.9750
Epoch 3/5
100/100 - 5s - loss: 0.0047 - accuracy: 1.0000 - val_loss: 0.0465 - val_accuracy: 0.9800
Epoch 4/5
100/100 - 6s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.0440 - val_accuracy: 0.9800
Epoch 5/5
100/100 - 6s - loss: 0.0015 - accuracy: 1.0000 - val_loss: 0.0426 - val_accuracy: 0.9800
<tensorflow.python.keras.callbacks.History at 0x7f93800f4210>
```

As can be seen from the epoch-wise performance feedback provided above, the model reaches an accuracy of 100% on the train set and 98% on the validation set.

We then test the model as shown below.

```
y_pred_ = model.predict(ts_datagen)

y_pred = y_pred_[:, 1] #---- USING ONLY PROBAILITIES FOR CLASS 1.
y_pred_labels = np.array([1 if i >= 0.5 else 0 for i in y_pred])
y_true = ts_datagen.classes

acc = (len(y_true)-abs(y_true - y_pred_labels).sum())/len(y_true)
acc
```

```
0.9801980198019802
```

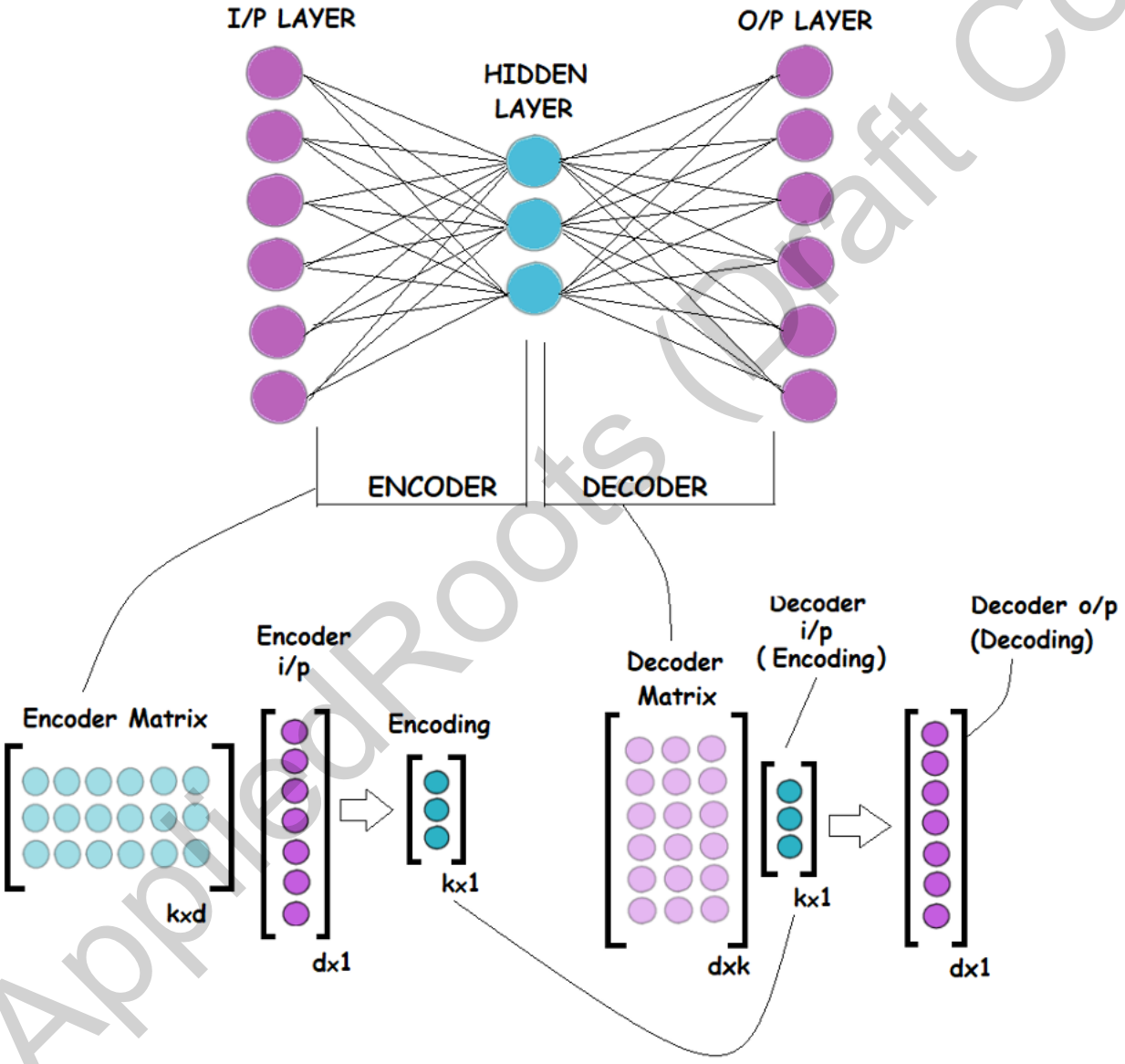
As can be seen from the output above, the model gives an accuracy of 98% on the test set.

Thus, by using a small amount of data and a fraction of the time (just 5 epochs) that would be needed to train a VGG model from scratch, we managed to get good performance by fine tuning pretrained models on new data. This is the core utility of Transfer Learning.

04. AUTOENCODERS

AUTOENCODERS

Autoencoders are a category of artificial neural networks used to perform efficient data encoding/compression. This is done using an encoder-decoder architecture. The image shown below describes a basic encoder-decoder architecture.



In the image shown above, the neural network consists of an input layer, a hidden layer and an output layer. The dimensionality d of the output layer is the same as the input layer, whereas the dimensionality k of the hidden layer is smaller than that of the input layer.

Now in conventional neural networks that perform classification, the output of the last layer is usually a vector whose dimensionality is the same as the number of classes in the dataset and

the corresponding label y is a one hot encoded vector having the same size, representing a particular class. In the case of autoencoders the labels are the input vectors themselves. This means that the weights (the encoder and decoder matrices shown above) learnt by such a neural network will be optimized to reconstruct the input as best as possible.

UNSUPERVISED LEARNING (DIMENSIONALITY REDUCTION):

Since autoencoders have no need for labels, they can be classified as an unsupervised learning method. The autoencoder's neural network can be broken down into two components:

THE ENCODER: This part of the network basically reduces the dimensionality of the inputs. It can have more than one hidden layer. The dimensionally reduced output of the encoder is called the encoding.

THE DECODER: This part of the network takes the output of the encoder and tries to recreate the original inputs. The decoder generally consists of the same layers as in the encoder, but in the reverse sequence.

Once an autoencoder has been trained on a dataset such that the loss observed between the input and the output (also known as reconstruction error) has been minimized, the decoder is then removed and the encoder is used to produce dimensionally reduced embeddings of unseen inputs of the same type.

LINEAR AND NON LINEAR ACTIVATIONS:

An interesting aspect about autoencoders is that, if unlike in conventional neural networks, one were not to use any nonlinear activations at each layer, the dimensionality reduced embedding derived from such a network would closely resemble that produced by Principal Component Analysis (PCA).

Due to the use of non linear activations, autoencoders are capable of producing complex non linear mappings between the input and the encoding, thus resulting in the capacity to reduce the dimensionality of the encoding even further than what PCAs are capable of.

UNDERCOMPLETE AND OVERCOMPLETE AUTOENCODERS:

The autoencoders discussed until now all had their encoding size (encoder output size) smaller than that of the input, thus resulting in dimensionality reduction. In other words the "bottleneck" imposed by the smaller size of the middle most hidden layer, forces the model to learn compressed representations of the input data. Under complete encoders are the most common type of autoencoders.

Overcomplete autoencoders, as the name suggests, produce encodings that have larger sizes than their inputs. This is due to the fact that their middle hidden layer has more number of nodes than the dimensionality of the input. Overcomplete autocoders simply represent a type

of autoencoder, but they have no significant applications.

AUTOENCODERS ON MNIST DATA:

For the sake of demonstration, we shall use the MNIST dataset and train/test various types of autoencoders to create embeddings of its images of numbers.

We download the MNIST dataset as shown below:

```
from keras.datasets import mnist

(X_tr, _), (X_ts, _) = mnist.load_data()

X_tr.shape, X_ts.shape
```

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

We then normalize each pixel value to be between 0 & 1 and then flatten each of the 28x28 sized images as shown below:

```
X_tr = X_tr.astype('float32') / 255.
X_ts = X_ts.astype('float32') / 255.

X_tr_ = X_tr.reshape(len(X_tr), 28*28)
X_ts_ = X_ts.reshape(len(X_ts), 28*28)

X_tr_.shape, X_ts_.shape
```

```
((60000, 784), (10000, 784))
```

We then create an autoencoder model as shown below, using the functional API that Keras provides. This API produces a directed acyclic graph of the layers. The format is quite similar to that of the sequential API and it could be easily understood by looking at the code shown below.

```
input_img = keras.Input(shape=(784,))
x = Dense(128, activation='relu')(input_img)
x = Dense(64, activation='relu')(x)

encoding = Dense(32, activation='relu')(x)

x = Dense(64, activation='relu')(encoding)
x = Dense(128, activation='relu')(x)

decoding = Dense(784)(x)

autoencoder_1 = keras.Model(input_img, decoding, name = 'basic_AE')
autoencoder_1.summary()
```

```
Model: "basic_AE"
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_2 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_3 (Dense)              (None, 64)                2112
_________________________________________________________________
dense_4 (Dense)              (None, 128)               8320
_________________________________________________________________
dense_5 (Dense)              (None, 784)               101136
=================================================================
Total params: 222,384
Trainable params: 222,384
Non-trainable params: 0
_________________________________________________________________
```

Note how the output of each layer is assigned to a variable and how the inputs to the layers are "fed" at the end of the layer definition within a curved bracket. The above structure creates a graph that links outputs from one layer to another all the way to the final output.

Note that in the last layer we have used the sigmoid activation function. Thus each of the values of the last layers are squashed to be in the [0, 1] range. This works well because the inputs themselves were normalized to be within that range.

The model is finally defined by using the "model" class and inputting the names of the input variable and the final output variable within it.

We then compile and train the model as shown below.

```
1  autoencoder_1.compile(optimizer='adamax', loss='binary_crossentropy')
2
3  autoencoder_1.fit(X_tr_, X_tr_, epochs=10, batch_size=100,
4                    shuffle=True, validation_data=(X_ts_, X_ts_),
5                    verbose = 1)
```

```
Epoch 1/10
600/600 [==============================] - 4s 3ms/step - loss: 0.3050 - val_loss: 0.1682
Epoch 2/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1611 - val_loss: 0.1410
Epoch 3/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1387 - val_loss: 0.1285
Epoch 4/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1284 - val_loss: 0.1218
```

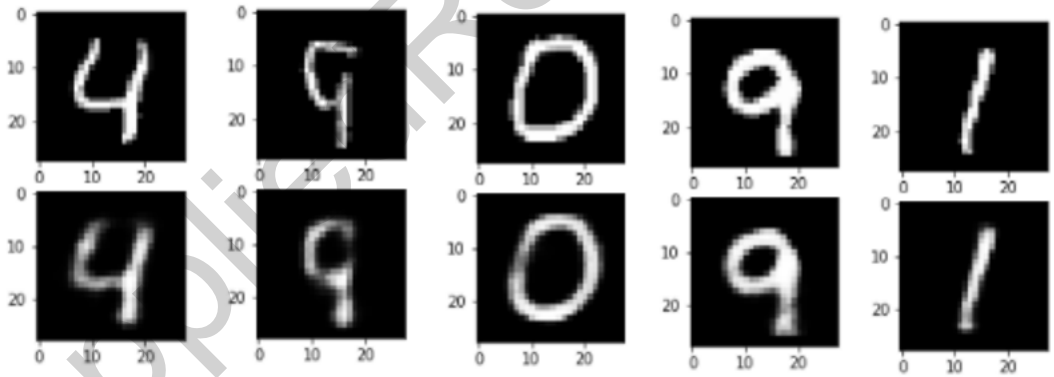
```
Epoch 5/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1220 - val_loss: 0.1161
Epoch 6/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1166 - val_loss: 0.1113
Epoch 7/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1116 - val_loss: 0.1084
Epoch 8/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1093 - val_loss: 0.1061
Epoch 9/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1070 - val_loss: 0.1044
Epoch 10/10
600/600 [==============================] - 2s 3ms/step - loss: 0.1052 - val_loss: 0.1026
<tensorflow.python.keras.callbacks.History at 0x7f86a8070cd0>
```

Note that we have used binary cross entropy as our loss function. Thus the loss is computed across each pixel and then aggregated for each input.

To check the efficacy of the encodings we can visually check how well the network reconstructs the images. If the reconstruction is good, then it implies that the dimensionally reduced encodings are of good quality.

We check the reconstructed output image as shown below.

```
1  X = X_ts
2  X_pred = autoencoder_1.predict(X_ts_).reshape(X_ts.shape)
3
4  fn_compare_images(X, X_pred)
```



The lower row represents the reconstructed images, whereas the represents the actual inputs.

As can be seen from the above output the autoencoder reconstructs the images quite well.

CONVOLUTIONAL AUTOENCODERS:

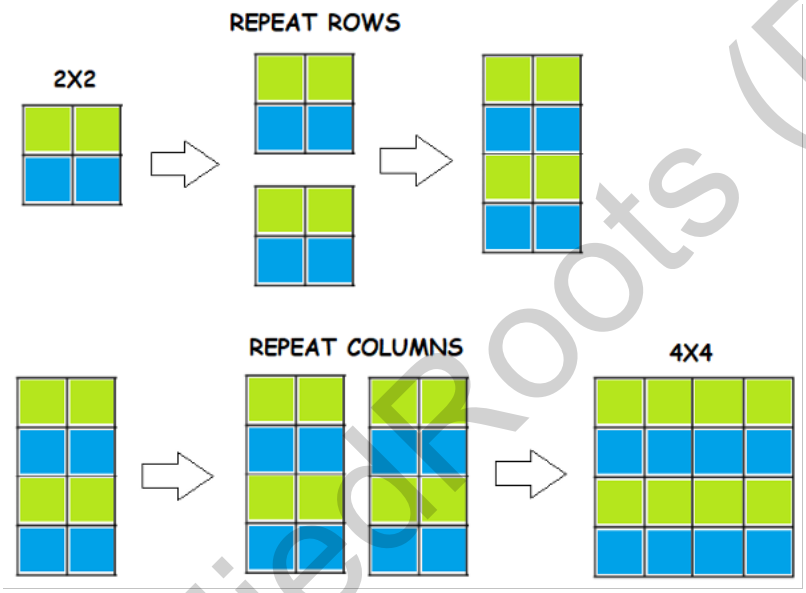
Convolutional autoencoders are autoencoders which are specifically designed to encode images. They use convolutional layers and max pooling in their encoder to reduce the dimensionality of the successive feature maps until the middle layer. The decoder uses a

combination of convolutional layers and upsampling layers to successively create larger feature maps from the feature map outputted by the middle layer to finally getting images of the same size as the inputs.

The encoder part of convolutional autoencoders are the same as the CNNS described till now. The decoder on the other hand creates larger feature maps from smaller ones. It does so by using upsampling layers.

UPSAMPLING LAYERS:

Upsampling layers do not contain trainable weights. They simply repeat each row in the input feature maps once thus resulting in a feature map with more rows. This resulting array is then subjected to repetition of each of its columns once. Thus finally resulting in an upsampled feature map with larger dimensions. The figure shown below expresses the process just described. A 2x2 feature map is upsampled to a 4x4 sized feature map.



Shown below is the code for implementing convolutional autoencoders for MNIST data. We first reshape the data to 3D inputs (n_images x length x breadth x n_channels).

```
X_tr = X_tr.reshape(*X_tr.shape, 1)
X_ts = X_ts.reshape(*X_ts.shape, 1)

X_tr.shape, X_ts.shape
```

```
((60000, 28, 28, 1), (10000, 28, 28, 1))
```

We then define our convolutional auto encoders as shown below. Note that we encode the 784 (28x28) dimensional input image vectors to 245 dimensions.

```
input_img = keras.Input(shape=(28, 28, 1))

sigmoid = dict(activation='sigmoid')
relu = dict(activation='relu')
padding= dict(padding='same')

x = Conv2D(10, (5, 5), **relu, **padding)(input_img)
x = MaxPooling2D((2, 2), **padding)(x)
x = Conv2D(5, (2, 2), **relu, **padding)(x)

encoding = MaxPooling2D((2, 2), **padding, name = 'ENCODING')(x)

x = UpSampling2D((2, 2))(encoding)
x = Conv2D(5, (2, 2), **relu, **padding)(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(10, (5, 5), **relu, **padding)(x) #-- No Padding

decoding = Conv2D(1, (3, 3), **sigmoid, **padding)(x)

autoencoder_2 = keras.Model(input_img, decoding, name = 'conv_AE_1
autoencoder_2.summary()
```

```
Model: "conv_AE_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 28, 28, 10)        260
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 14, 14, 10)        0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 14, 14, 5)         205
_________________________________________________________________
ENCODING (MaxPooling2D)      (None, 7, 7, 5)           0
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 14, 14, 5)         0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 14, 14, 5)         105
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 28, 28, 5)         0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 28, 28, 10)        1260
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 28, 28, 1)         91
=================================================================
Total params: 1,921
Trainable params: 1,921
Non-trainable params: 0
_________________________________________________________________
```

7x7x5 = 245

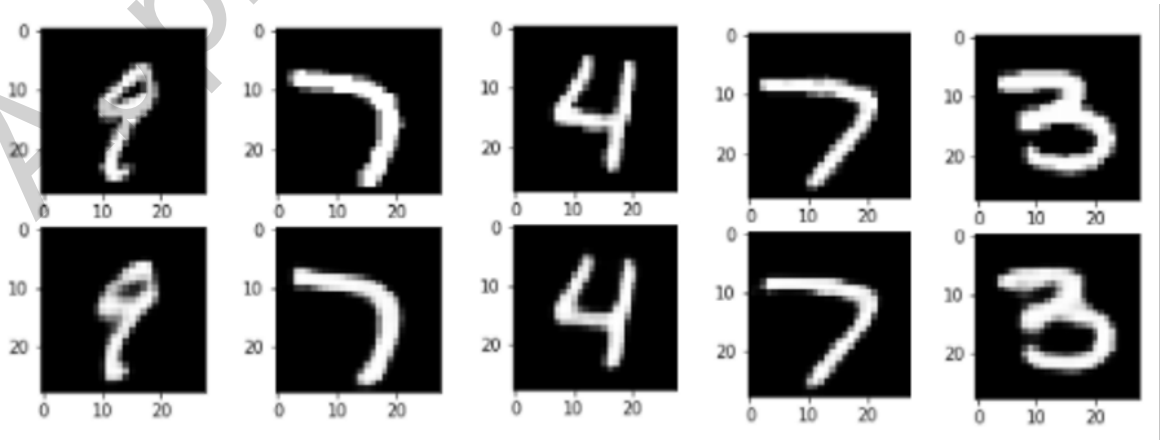
We then train the above model as shown below.

```
autoencoder_2.compile(optimizer='adamax', loss='binary_crossentropy')

autoencoder_2.fit(X_tr, X_tr, epochs=10, batch_size=100,
                  shuffle=True, validation_data=(X_ts, X_ts),
                  verbose = 1)
```

```
Epoch 1/10
600/600 [==============================] - 3s 5ms/step - loss: 0.2955 - val_loss: 0.1015
Epoch 2/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0988 - val_loss: 0.0905
Epoch 3/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0901 - val_loss: 0.0860
Epoch 4/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0860 - val_loss: 0.0828
Epoch 5/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0834 - val_loss: 0.0809
Epoch 6/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0814 - val_loss: 0.0797
Epoch 7/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0803 - val_loss: 0.0788
Epoch 8/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0795 - val_loss: 0.0781
Epoch 9/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0788 - val_loss: 0.0775
Epoch 10/10
600/600 [==============================] - 3s 4ms/step - loss: 0.0781 - val_loss: 0.0770
<tensorflow.python.keras.callbacks.History at 0x7f480a044e50>
```

To check the efficacy of the encodings, we visually check how well the network reconstructs the images as shown below.

```
X = X_ts
X_pred = autoencoder_2.predict(X_ts)

fn_compare_images(X, X_pred)
```



As can be seen from the above output the autoencoder reconstructs the images quite well.

DENOISING AUTOENCODERS:

So far we discussed the concept of training an autoencoder where the input and outputs are identical and the model is tasked with reproducing the input as closely as possible, while passing through an information bottleneck (sequential reductions in the number of dimensions of the input data). In general, we would like the autoencoder to be sensitive enough to recreate the original observation but insensitive enough to the training data such that the model learns a generalizable encoding. An approach towards developing a generalizable model is to slightly corrupt the input data but still maintain the uncorrupted data as our target output. This is done as shown below.

```
def fn_add_noise(data, noise_level = 0.5):
    dist = np.random.normal(loc=0.0, scale=1.0, size=data.shape)
    noise = noise_level * dist
    noisy_data = np.clip(data+noise, 0., 1.)
    return noisy_data

```

```
X_tr_noisy = fn_add_noise(X_tr)
X_ts_noisy = fn_add_noise(X_ts)
```

We then define an autoencoder model with the same specifications as before.

```
input_img = keras.Input(shape=(28, 28, 1))

sigmoid = dict(activation='sigmoid')
relu = dict(activation='relu')
padding= dict(padding='same')

x = Conv2D(10, (5, 5), **relu, **padding)(input_img)
x = MaxPooling2D((2, 2), **padding)(x)
x = Conv2D(5, (2, 2), **relu, **padding)(x)

encoding = MaxPooling2D((2, 2), **padding, name = 'ENCODING')(x)

x = UpSampling2D((2, 2))(encoding)
x = Conv2D(5, (2, 2), **relu, **padding)(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(10, (5, 5), **relu, **padding)(x) #-- No Padding

decoding = Conv2D(1, (3, 3), **sigmoid, **padding)(x)

autoencoder_3 = keras.Model(input_img, decoding, name = 'conv_AE_2')
```

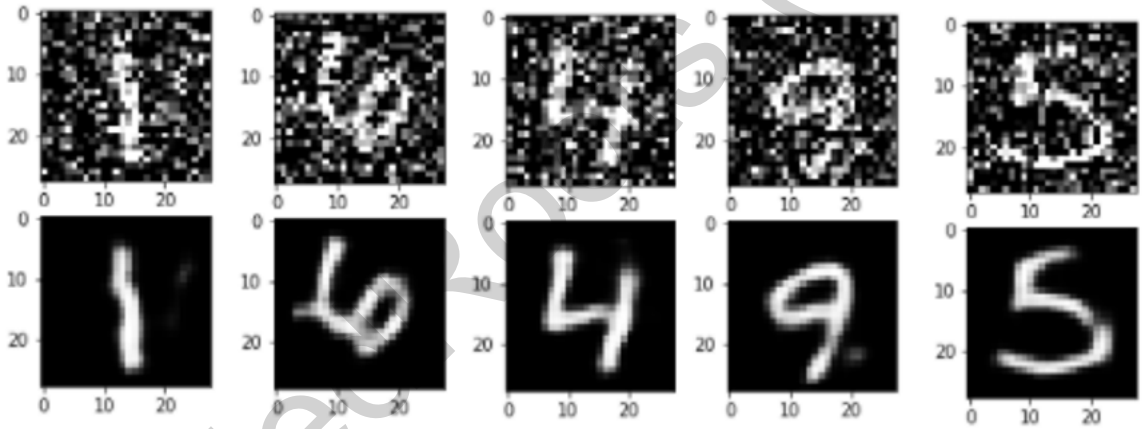
We then train the above model on the noisy data as shown below. Note how only the input data is noisy, but the target/label data is the original non noisy data.

```
autoencoder_3.compile(optimizer='adamax', loss='binary_crossentropy')

autoencoder_3.fit(X_tr_noisy, X_tr, epochs=50, batch_size=100,
                  shuffle=True, validation_data=(X_ts_noisy, X_ts),
                  verbose = 0)
```

```
<tensorflow.python.keras.callbacks.History at 0x7f4756e086d0>
```

We then visually check how well the network reconstructs the images as shown below.

```
X = X_ts_noisy
X_pred = autoencoder_3.predict(X_ts_noisy)

fn_compare_images(X, X_pred)
```



As can be seen from the output above, the autoencoder has managed to reconstruct the images clearly despite the noise in the inputs. This means that we have produced a model that will generalize well to images outside the training data and also has “noise resilience” baked into it.

As can be understood from the images above, autoencoders can also be used for image reconstruction tasks.

05. GENERATIVE ADVERSARIAL NETWORKS

GENERATIVE ADVERSARIAL NETWORKS

Generative Adversarial Networks (GANs) are a approach to generative modeling using neural networks. Generative models in essence, try to estimate/model the distribution of the data they are exposed to, so that new data of the same kind can then be sampled from this modelled distribution.

Generative Adversarial Networks were basically designed to be able to generate realistic images by learning the **latent space** of the image they are trained on. We shall discuss what this means as the chapter unfolds.

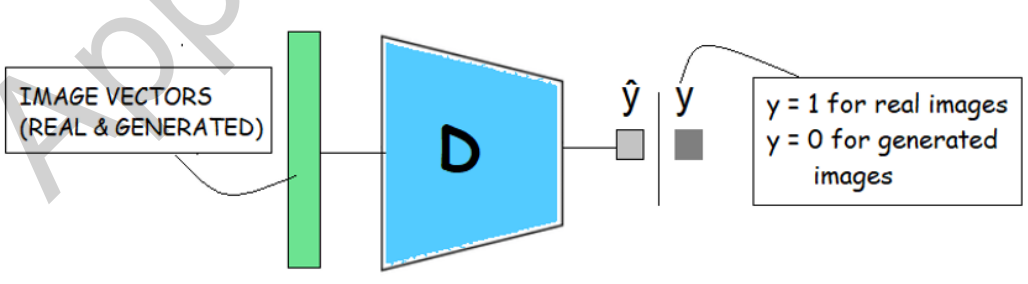
Generative Adversarial Networks use a very unique method, where the task is framed as a supervised learning problem involving two sub-models called:

1. The **generator** and
2. The **discriminator**

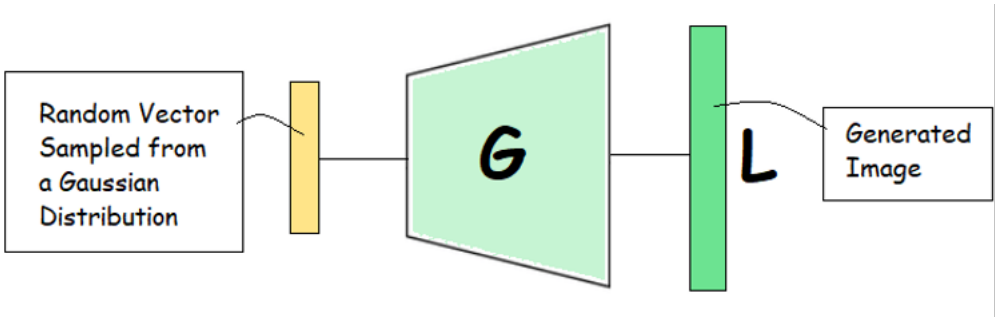
The generator's task is to produce/generate images such that they capture the characteristics of the training dataset, so much so that the samples it generates look in distinguishable from the training data, whereas the discriminator's task is to discern between images that are real (ie: originating from the train set) and those that were created by the generator. The two models are set up in an adversarial mode, such that during training each of the models keeps getting better at their specific tasks by trying to outperform each other.

FUNDAMENTAL ARCHITECTURE OF GANS AND ITS TRAINING CYCLE:

As discussed previously, GANs are a composite of two neural network models. The discriminator (D) is basically a binary classifier which receives real and generated images as its inputs, where the real images are labelled '1' and the fake images are labelled '0'.



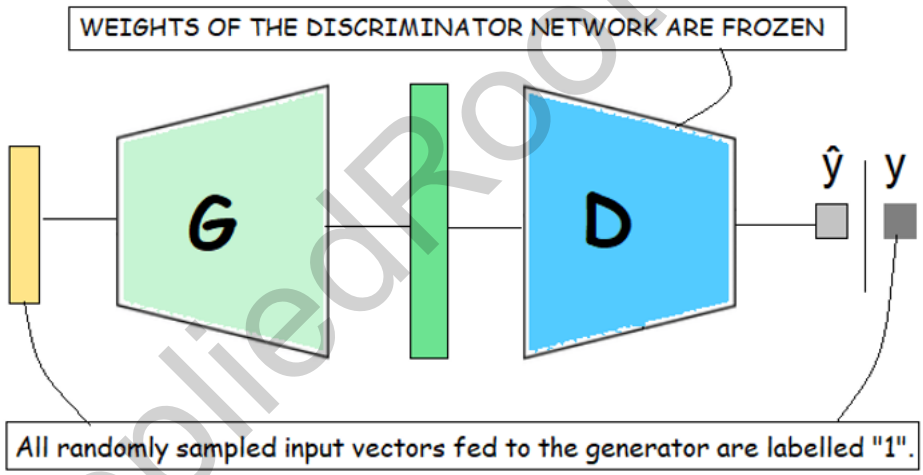
The Generator (G) is the same as the decoder of an autoencoder. The only difference here is that instead of "decoding" the encoder's output back into the original input image, the generator takes in as its input random vectors of some arbitrary fixed size, sampled from a gaussian distribution and tries to transform them into images.



The generator and discriminator are trained alternately, where first the generator generates image vectors using the random vectors it receives as its input. These initial image vectors are generated based on the randomly initialized untrained weights of the generator's neural network.

The discriminator is then trained on actual images that we want the GAN to emulate (labeled 1) and the image vectors generated by the generator (labelled 0). After training, the discriminator becomes capable of classifying actual images and generated images.

Now, just like in the case of Transfer Learning, we freeze the weights of the discriminator's network and combine the generator and the discriminator, as shown in the image shown below.



The generator is then trained using this configuration. The important detail to notice here is - that all the labels for the randomly sampled input vectors are flipped to 1 instead of 0. Due to this, the generator is forced to learn a weight configuration for its neural network that would produce images (L), which when passed through the frozen weights of the discriminator results in $\hat{y} = 1$ as much as possible (ie: produce least loss). Note that the weights of the discriminator had been previously trained to discern between generated and real images.

The training cycle of GANS can be summarized as outlined below:

1. The generator is made to generate a batch of images in response to being fed a batch of input vectors of some arbitrary fixed size randomly sampled from a gaussian distribution.
2. We randomly sample a batch (of the same size as above), of real images from the training data.
3. We train the discriminator to discern between real images(labelled 1) and generated images (labelled 0) using the data collected in steps 1 and 2.
4. We then train the generator in combination with the trained discriminator, but with its weights frozen. We do this by using the "combined model" and feeding the generator of the combined model with a randomly sampled batch of input vectors just like described in step 1, but this time with their corresponding labels flipped to 1.

Steps 1 to 4 are repeated for a suitable number of epochs till the images outputted by the generator seem to look much like the real images.

In the steps outlined above, the loss of the combined model (while training the generator) reduces if the generator is able to generate images which the discriminator classifies as "real". Since the weights of the generator are learnt based on the feedback it received from the loss function it can be said that the generator learns through the feedback it receives from the discriminator's classifications.

So at every iteration, both the generator and the discriminator improve. The generator improves because it uses the feedback received from the discriminator to update its weights and so it starts generating more realistic images. The discriminator improves as well, because as the iterations proceed, the generator keeps generating images that are generally better than the ones it did in the previous cycle. So at every iteration the discriminator has to learn to discriminate between real images and generated images, where the generated images are continuously getting more realistic. Thus both networks continue to improve simultaneously by being engaged in an adversarial relationship.

GAN FOR GENERATING HANDWRITTEN DIGITS:

For the sake of demonstration we shall implement a basic GAN model for generating images of handwritten digits by training it on MNISTdata.

We use keras' inbuilt MNIST dataset:

```
from keras.datasets import mnist

(X_tr, _), (X_ts, _) = mnist.load_data()
X_tr.shape, X_ts.shape
```

```
Downloading data
11493376/11490434 [==============================]
((60000, 28, 28), (10000, 28, 28))
```

We then scale all the image pixel values to be within -1 and 1:

```
X_tr = (X_tr.astype('float32') - 127.5)/127.5
X_ts = (X_ts.astype('float32') - 127.5)/127.5

X_tr.min(), X_tr.max()
```

```
(-1.0, 1.0)
```

For GANs it has been empirically found that scaling values to be in the range [-1, 1] works better than scaling them to be in the interval [0, 1] as done previously.

We then flatten all the 2D images into 1D vectors as shown below:

```
X_tr = X_tr.reshape(len(X_tr), 28*28)
X_ts = X_ts.reshape(len(X_ts), 28*28)

X_tr.shape, X_tr.shape,
```

```
((60000, 784), (60000, 784))
```

We then define functions that return generator and discriminator models as shown below:

```
def fn_generator():

    model = Sequential()

    model.add(Dense(units=256,input_dim=100))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.7))

    model.add(Dense(units=512))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.7))

```

```
    model.add(Dense(units=1024))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.7))
    model.add(Dense(units=784, activation='tanh'))

    return model
```

```
def fn_discriminator():

    model = Sequential()

    model.add(Dense(units=512, input_dim=784))
    model.add(LeakyReLU(0.2))

    model.add(Dense(units=256))
    model.add(LeakyReLU(0.2))

    model.add(Dense(units=1, activation='sigmoid'))
    model.compile(loss = 'binary_crossentropy',
                  optimizer = Adam(lr=0.0002, beta_1=0.5))
    return model
```

Note how in the functions above only the discriminator model is compiled with a loss function and optimizer, the generator model lacks these.We then create our generator and discriminator models as shown below:

```
GENERATOR = fn_generator()
DISCRIMINATOR = fn_discriminator()
```

We inspect the create models as shown below:

```
GENERATOR.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 256)               25856
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 256)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 256)               1024
_________________________________________________________________
dense_1 (Dense)              (None, 512)               131584
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
```

```
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              525312
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
batch_normalization_2 (Batch (None, 1024)              4096
_________________________________________________________________
dense_3 (Dense)              (None, 784)               803600
=================================================================
Total params: 1,493,520
Trainable params: 1,489,936
Non-trainable params: 3,584
_________________________________________________________________
```

```
1  DISCRIMINATOR.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               401920
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               131328
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 257
=================================================================
Total params: 533,505
Trainable params: 533,505
Non-trainable params: 0
_________________________________________________________________
```

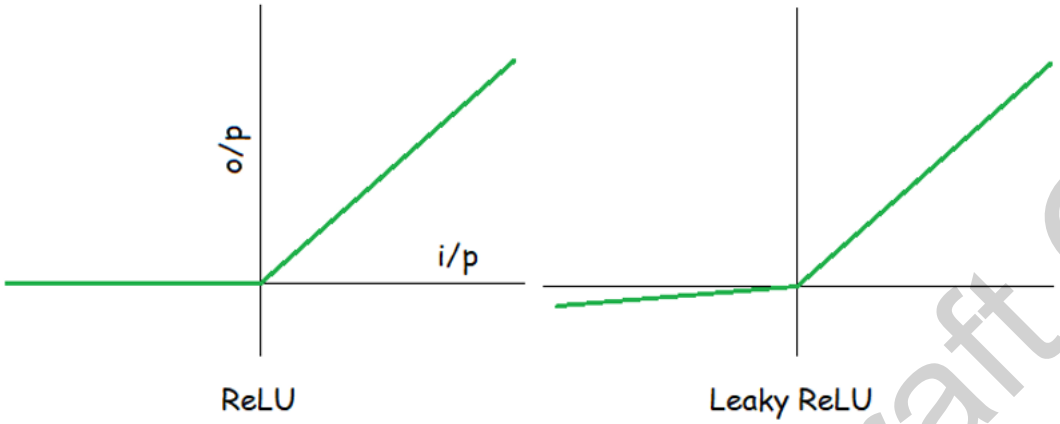
GAN models are known to be notoriously difficult to train and even slight changes in the models training parameters can make a big difference in the quality of the generated images. Most of the parameter settings used in GAN models above are heuristic in nature, which have been arrived at by hacking and experimentation rather than scientific formalism. The mathematics behind the workings of GANs are quite involved and not yet completely formalized.

Leaky ReLU is preferred instead of the conventional ReLU activation function in the case of GANs. Leaky RELUs are similar to conventional ReLU, the only difference is that instead of the function being y = max(0, x), it is defined as:

y = max(alpha*x, x)

Here the parameter alpha is called the negative slope coefficient. It is a very small value in the range of 0 to 1. Leaky ReLU allows the neurons to activate even if the input values are negative by outputting a very small negative value that is proportional to the negative input. Thus the

neuron learns even in the case of negative activations since Leaky ReLUs have a (very small) slope even in the negative region. In the case of the discriminator and generator models being implemented, alpha is set to 0.2.



Also note the usage of batch normalization in the generator, with its momentum parameter set to 0.2. The momentum parameter in case of the batch normalization layers basically performs a similar function as it does in the case of the optimizers. The mean and standard deviation values used in the computation of batch normalization (see chapter: Deep Learning) are modified by the "momentum" provided by the means and standard deviations computed in the previous training iterations. As shown in the summary above we use a momentum of 0.7 in the generator's batch normalization layers.

We then define a function for creating the "combined model" as shown below:

```
def fn_COMBINED_MODEL(GENERATOR, DISCRIMINATOR, latent_input_dims = 100):

    DISCRIMINATOR.trainable = False

    latent_input = Input(shape = (latent_input_dims,))
    x = GENERATOR(latent_input)
    pred = DISCRIMINATOR(x)
    model = Model(latent_input, pred)
    model.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))

    return model
```

We create the combined model a shown below:

```
COMBINED_MODEL = fn_COMBINED_MODEL(GENERATOR, DISCRIMINATOR)
COMBINED_MODEL.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 100)]             0
_________________________________________________________________
sequential (Sequential)      (None, 784)               1493520
_________________________________________________________________
sequential_1 (Sequential)    (None, 1)                 533505
=================================================================
Total params: 2,027,025
Trainable params: 1,489,936
Non-trainable params: 537,089
_________________________________________________________________
```

Note that the layers of the discriminator have trainable and non-trainable layers. The non-trainable layers belong to the discriminator. They were frozen while creating the combined model using the function shown above. **Thus the weights of the discriminator are frozen when used within the combined model, but are trainable when used separately by itself outside the combined model.**

We then define a function which we used the train the GAN as discussed previously:

```
def fn_train_GAN(latent_dim, batch_size, n_epochs):

    ones = np.ones(batch_size)
    zeros = np.zeros(batch_size)
    d_losses, g_losses = [], []

    pbar = ProgressBar()
    for i in pbar(range(n_epochs)):

        # SELECT A RANDOM BATCH OF IMAGES
        idx = np.random.choice(range(len(X_tr)), batch_size)
        real_imgs = X_tr[idx]

        # GENERATE IMAGES
        noise = np.random.randn(batch_size, latent_dim)
        generated_images = GENERATOR.predict(noise)

        # TRAIN THE DISCRIMINATOR
        d_loss_real  = DISCRIMINATOR.train_on_batch(real_imgs, ones)
        d_loss_fake = DISCRIMINATOR.train_on_batch(generated_images, zeros)
        d_loss = 0.5 * (d_loss_real + d_loss_fake)
```

```
22
23          #TRAIN THE GENERATOR(ie:COMBINED_MODEL) TWICE
24          noise = np.random.randn(batch_size, latent_dim)
25          g_loss = COMBINED_MODEL.train_on_batch(noise, ones) #---- labels are ones
26          noise = np.random.randn(batch_size, latent_dim)
27          gan_loss = COMBINED_MODEL.train_on_batch(noise, ones) #-- labels are ones
28
29          # SAVE LOSSES
30          d_losses.append(d_loss)
31          g_losses.append(g_loss)
32
33      return GENERATOR, d_losses, g_losses
```

Note that in the function above we use the **train_on_batch** method available for keras' model class. This method allows us to run a single gradient update on a single batch of data. Also, the discriminator is trained on real images and generated images in two separate steps. This is the preferred technique for training the discriminator.

The loss of the discriminator is computed by averaging the the loss produced by the model during these two training steps.

We then train the GAN for 5000 epochs as shown below:

```
1  latent_dim = 100
2  batch_size, n_epochs = 50, 5000
3
4  TRAINED_GENERATOR, d_losses, g_losses = fn_train_GAN(latent_dim, batch_size, n_epochs)
```

```
100% (5000 of 5000) |####################| Elapsed Time: 0:06:07 Time:  0:06:07
```

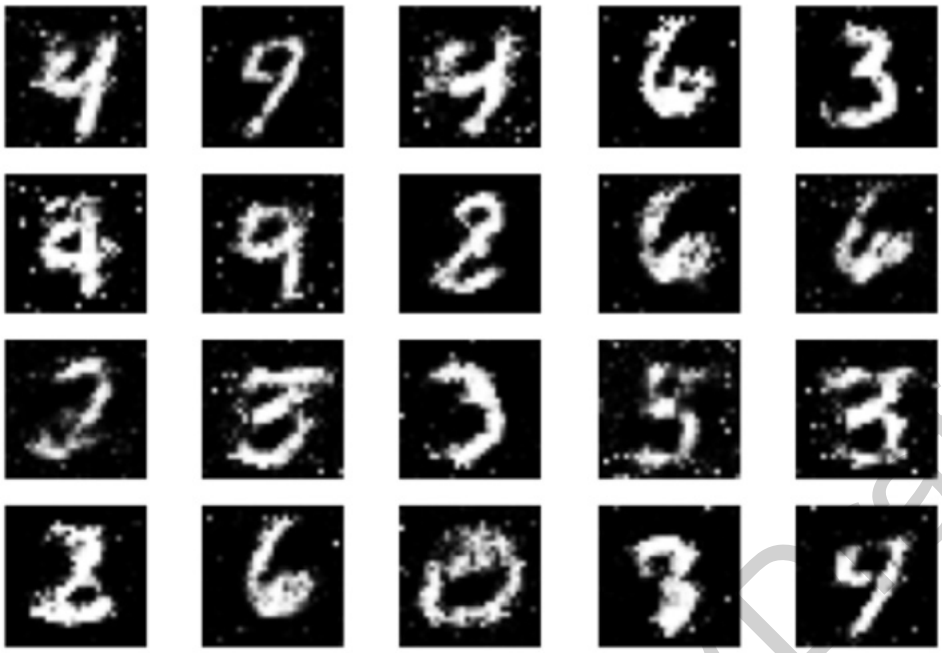
The **latent_dim** parameter of the function shown above represents the dimensionality of the input vectors (to the generator) randomly selected from a gaussian distribution.

We then visually check the images generated by the GAN as shown below:

```
1  noise = np.random.randn(20, latent_dim)
2  generated_imgs = TRAINED_GENERATOR.predict(noise)
3  generated_imgs.shape
```

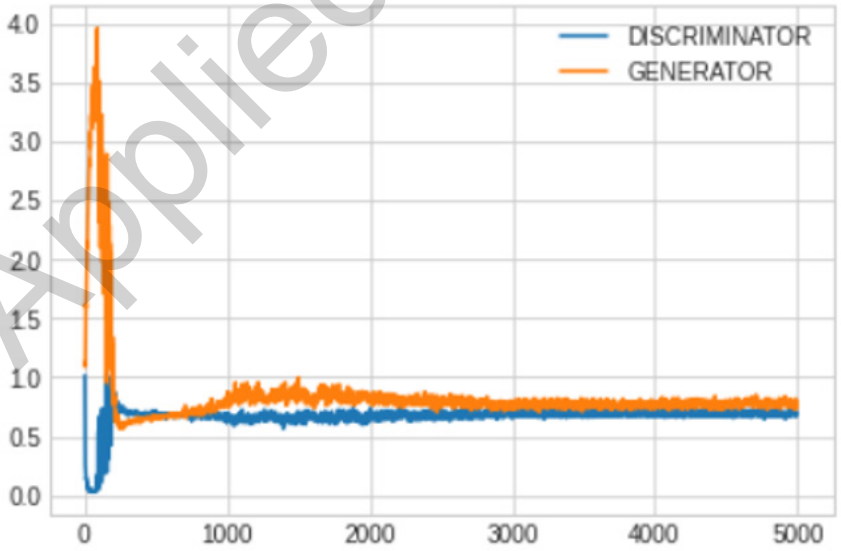
```
(20, 784)
```

```
1   fn_show_imgs(generated_imgs, imgs_per_row = 5)
```



We can inspect the performance of the generator and discriminator by inspecting their losses as shown below:

```
1   plt.plot(range(len(d_losses)), d_losses, label = 'DISCRIMINATOR')
2   plt.plot(range(len(d_losses)), g_losses, label = 'GENERATOR')
3   plt.legend()
4   plt.show()
```



As can be seen from the loss plot above, the loss of the discriminator is quite low in the beginning, whereas those of the generator are quite high. This is because the images generated by the generator in the beginning are of lower quality. Thus the discriminator is easily able to recognize them as "not real" quite easily, thus resulting lower losses. The generator loss (loss of the combined model) is higher for the same reason that the images fed to the discriminator by the generator do not result in the prediction being y = 1 for the first 100 to 300 iterations.

But as the iterations progress the generator starts generating more realistic images resulting in lower generator losses and higher discriminator losses until eventually they feed off each other and reach a kind of equilibrium state where both keep improving at the same rate. It is for this reason that the loss value of the GAN's sub models shown above are not directly indicative of their performance. The performance of the models keep improving despite the apparent static nature of the loss values, since each model is in an adversarial relationship with the other. In other words, as the generator improves its losses decrease, while that of the discriminator increases, but at the same time as the discriminator improves its loss decreases, while that of the generator increases.

LATENT INPUT SPACE OF A GAN:

From the discussions till now, we know that the generator model is basically responsible for creating new images that are passable as real. It does this by taking a point (100D point in the case of the example above) from an arbitrary latent space as its input and then outputting a corresponding grayscale image.

The latent space inherently has no meaning, but by randomly drawing points from this space and providing them to the generator during training, the generator assigns meaning to this latent space. By the end of its training, the vectors in this space become a compressed representation of the output space (ie:MNIST images), such that they can be 'decoded' by the generator to produce images like the real ones the GAN was trained on.

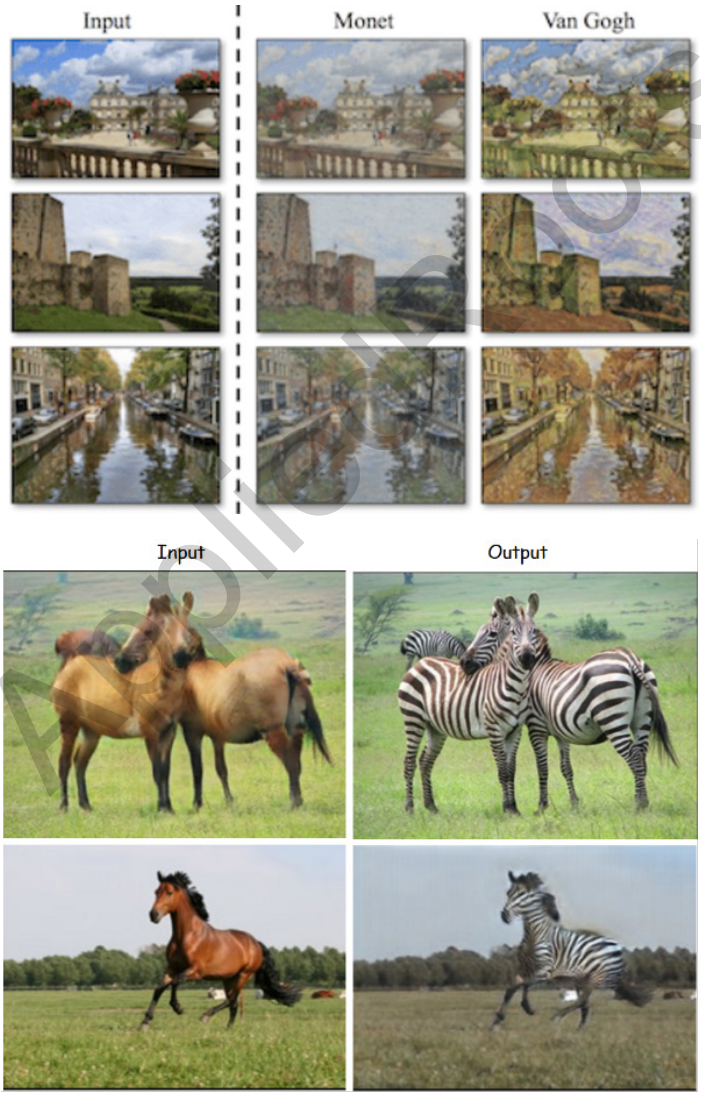
CAPABILITIES OF ADVANCED GANS:

Despite the inherent difficulties associated with training GANs, they have grown in complexity and sophistication since the time they were first introduced in 2014. The example demonstrated previously on the MNIST dataset is a simplified introduction to GANs. Most advanced GANs use CNNs instead of the fully connected layers in their generator and discriminator models.

GANs of the present day have the capacity to generate images that are completely realistic and indistinguishable from the real images they were trained on. A variant called the StyleGAN is capable of generating images of human faces that do not actually exist and are completely indistinguishable as not being real. None of the faces shown below are real, they were all

generated by a StyleGAN.

Many GANs have been developed in the past few years, each designed specifically for specialized tasks. For example the CycleGAn architecture is designed to perform domain transfer tasks like transforming raw images into the "style" of famous painters or transforming horse pictures to zebra pictures and the like. The images shown below demonstrate their capability.

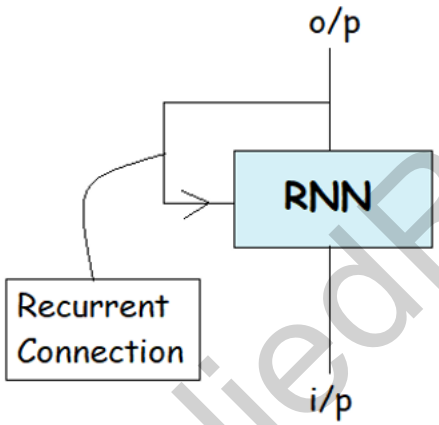


06. RECURRENT NEURAL NETWORKS

RECURRENT NEURAL NETWORKS

All the machine learning models that were used till now, on the amazon reviews dataset, did not take into account the sequence of the words in the reviews. But the fact is that language is sequential. The meaning of the sentence depends on the sequential order of the words it contains and the meaning of a paragraph depends on the sequential order of the sentences within it. Information derived from previous words/sentences/paragraphs help us understand the meaning of the current word/sentence/paragraph being read. In other words, the sequence itself contains information apart from the individual meaning the words may be having.

Recurrent neural networks have an architecture that is specially designed to learn sequential information from sequential data. These neural networks have "loops" within them that allow previous information to persist and have an effect on the information being derived from current data. The diagram shown below depicts the core essential idea of Recurrent neural networks.

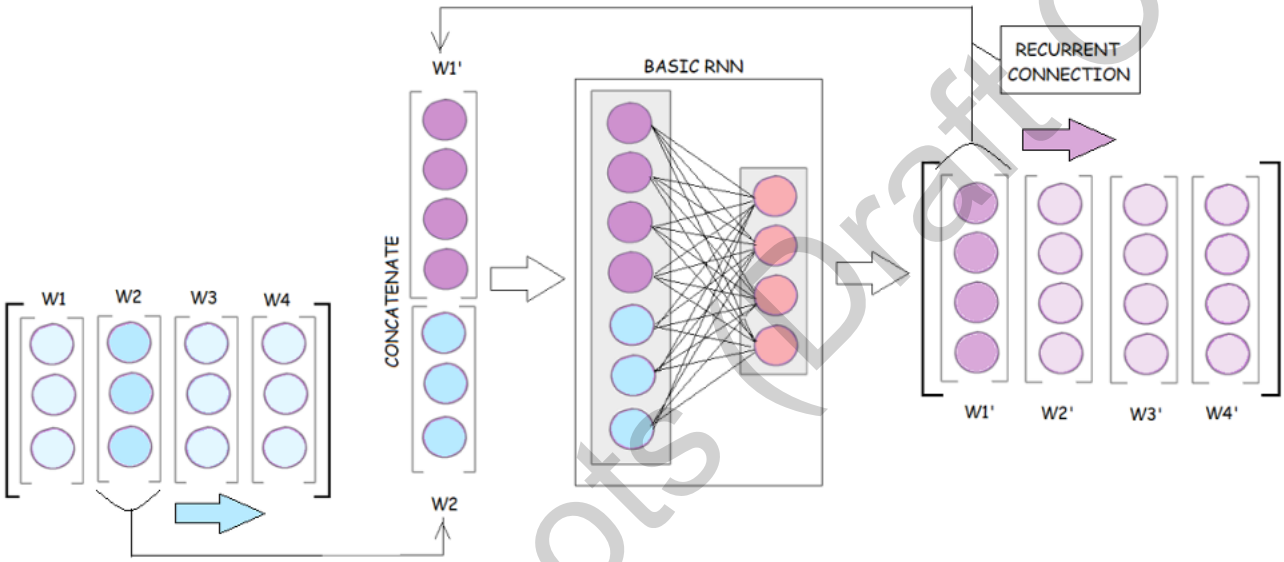


Recurrent neural networks process sequences by iterating through them one by one and maintaining a "hidden state" representing information about what has been processed thus far.

Consider the example of the amazon reviews sentiment classification task. Say that the RNN is currently processing some ith review/text doc in the dataset. The image below depicts the workings of a basic RNN, where the vector representing the second word that of that ith review is currently being iterated (ie: The first word has already passed through the RNN and produced a corresponding output).

Note how while processing the second word, the output of the first (previous) iteration is used as part of the input for the second iteration. This is the recurrence that happens in RNNs. The output of the previous iteration is used in two ways:

1. It becomes part of the predicted output sequence (ex: W1', W2', W3', W4' corresponding to W1, W2, W3, W4 as shown in the image below).
2. It becomes a part of the input of the next iteration (ex: W1' becomes the "**hidden state vector**" for word W2).



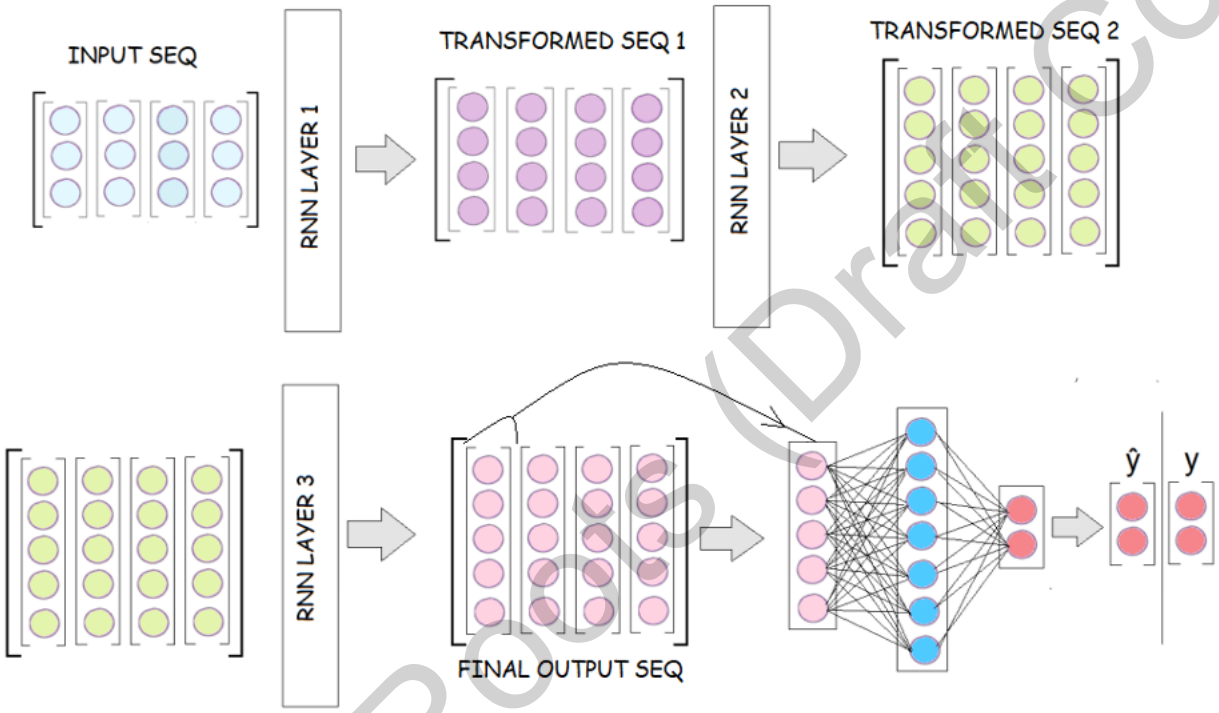
Note that for the first iteration (while processing W1), there is no recurrence so the "hidden state vector" will just be a zero vector having a size that is the same as the size chosen for the output vectors. The output size is determined by the number of nodes in the neural layer used within the RNN (4 for the example used in the image above).

TYPES OF RNN OUTPUTS:

As can be seen from the previous image, there are two options for how the output of a RNN can be used.

1. Sequential outputs: Here the entire sequence outputted by the RNN, given an input sequence, is considered as its output.
2. Non Sequential outputs: Here only the last vector in the output sequence is considered as its output.

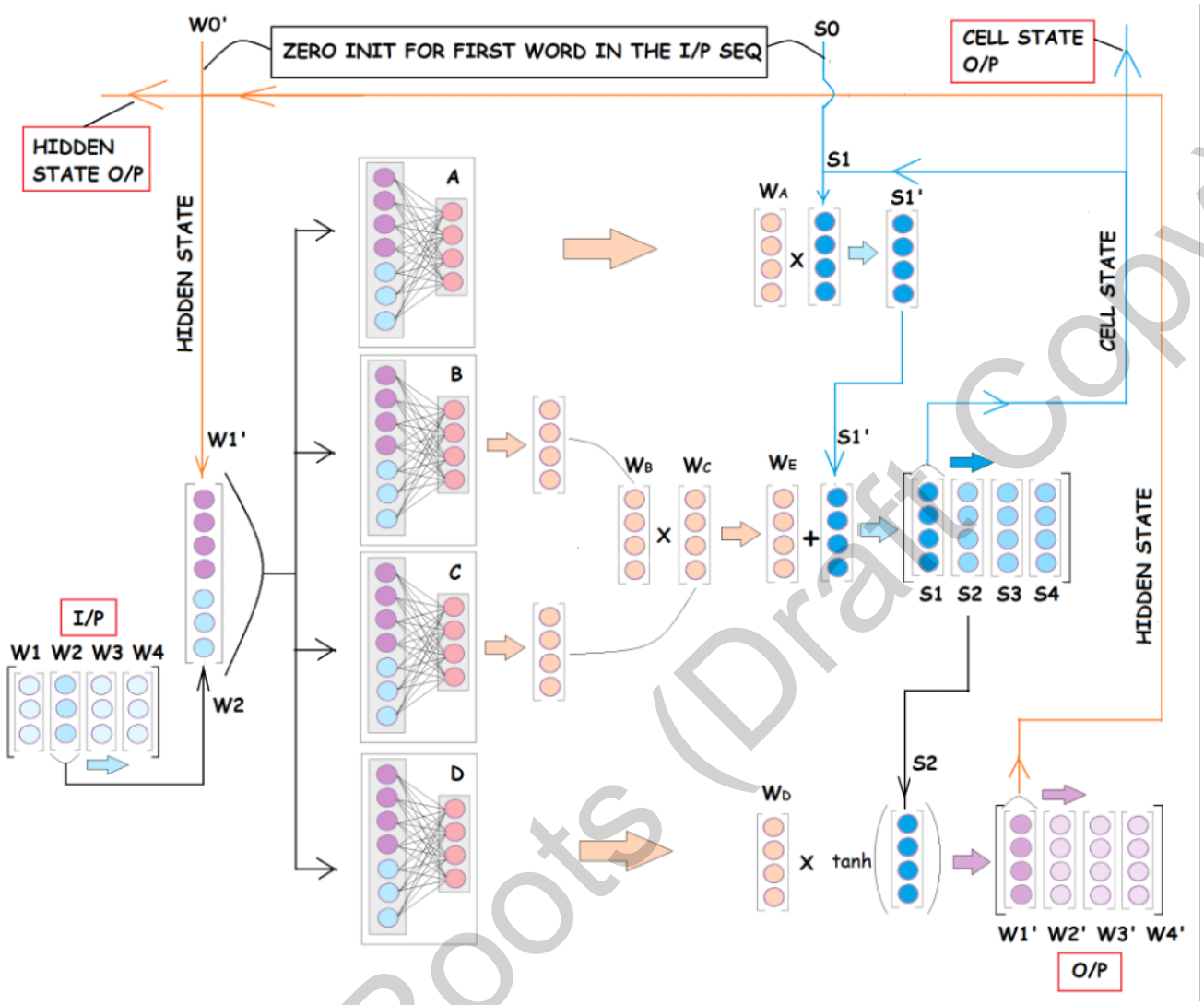
In stacked RNNs, multiple RNNs are used one after the other (just like in MLPs) the output of one RNN layer becomes the input for the next RNN and so on. In such cases for all the RNN layers except for the last one we use option 1 and for the last RNN we use option 2. This is expressed in the image shown below, for the last layer in a 3 layered stacked RNN, we do not take the entire sequence outputted by the RNN but just the last vector in the output sequence. This single vector then forms the input to a conventional multilayer neural network that finally outputs a vector or a scalar, which is compared to the labels (classification/regression).



LONG SHORT TERM MEMORY NETWORKS (LSTM):

The basic RNNs discussed above do not work well for long input sequences where the gap between relevant information and the point where it is needed can be large. In other words, in a long text sequence, it is quite often that a word currently being processed may need the co-reference of some word or phrase that occurred "a long time back" to make complete sense. Basic RNNs do not have the capacity to learn such long term dependencies.

Long Short Term Memory networks are a special kind of RNN specifically designed for learning long-term dependencies. Remembering information for long periods of time is built into their default behavior. LSTMs have a different structure compared to the basic RNN discussed previously. Instead of having a single neural network layer, there are four that interact in a very specific way. The image shown below describes the basic architecture of LSTMs.



The LSTM consists of four neural layers A, B, C & D. It has 3 inputs and outputs. A, B & D have sigmoid activations whereas C has tanh activation.

The image above is basically a "snapshot" of what is happening in the LSTM while it is processing the second word W2 in the input sequence (First word has already been processed).

Just as in the case of basic RNNs, the input vector currently being processed (W2) is concatenated with the previous output (hidden state) of the LSTM (ie: W1'). The resulting vector is fed to each of the neural layers A, B, C & D, which results in the vectors WA, WB, WC & WD respectively. Vectors WB & WC are then multiplied to get vector WE.

The internal process of the LSTM can be understood using the perspective of the following two types of vectors that interact within the it:

1. INFORMATION VECTORS: These are vectors whose value range is -1 to 1 and they are the outcome of a vector passing through a neural layer with a tanh activation (ie: WC corresponding to Neural layer C).

2. IMPORTANCE VECTORS: These are vectors whose value range is 0 to 1 and they are the outcome of a vector passing through a neural layer with a sigmoid activation (ie: WA, WB & WD corresponding to layers A, B & D).

Both these vectors have the same size and they occur in pairs. The values in the importance vector can be said to signify the importance of the corresponding values in the information vector it is paired with.Thus if a value in the information vector has a corresponding value in the importance vector that is close to one, then that value is considered important.

The multiplication of an information vector with an importance vector results in another vector that could be considered as containing "relevant information". The larger the value in the importance vector (closer to 1) the lesser the corresponding value in the information vector is changed (ie: decreased).

The key to LSTMs is the "**cell state**". In the image above the loop formed by the blue flow lines (top right side of the image) represent how the cell state is generated and used. The cell state or cell state vector represents the "long term" memory component of the LSTM. Old impertinent information is removed from it and new pertinent information is added to it at each step while processing sequential information. In the "snapshot" shown above, the cell state vector S1 corresponding to the previous word in the input sequence goes through the following two modifications:

1. MULTIPLICATION WITH VECTOR WA: This operation could be likened to the act of removing non-useful information from the cell state vector. In the "snapshot" being considered, the previous cell state vector S1 could be considered as an "information vector" representing previous memory. The multiplication of S1 with WA (which is an importance vector) produces a new cell state vector S1' containing more relevant "previous" information with respect to the current word being iterated.

2. ADDITION WITH VECTOR WE: This operation could be likened to the act of adding new useful information to S1'. Here WE is the outcome of the product of WB & WC where WB is an importance vector and WC is an information vector.

WC can be viewed as the vector containing "new information" derived from the current word being processed and WB can be seen as the vector which decides how much of the new information in WC is relevant. The product of WB & WC (ie: WE) can thus be seen as representing "new relevant information".

The addition of S1' with WE thus represents new relevant information being added to old relevant information, resulting in a new cell state S2 which represents new useful information.

The new cell state vector S2 is then used in two ways:

1. It becomes the cell state vector for the next word (ie: W3) in the input sequence.
2. It is used to create the output corresponding to the current word in the input sequence. This is done by passing the cell state vector S2 through a tanh function thus transforming into an information vector. This is then multiplied with vector WD which is an importance vector thus resulting in the output W2' corresponding to W2 in the input sequence.

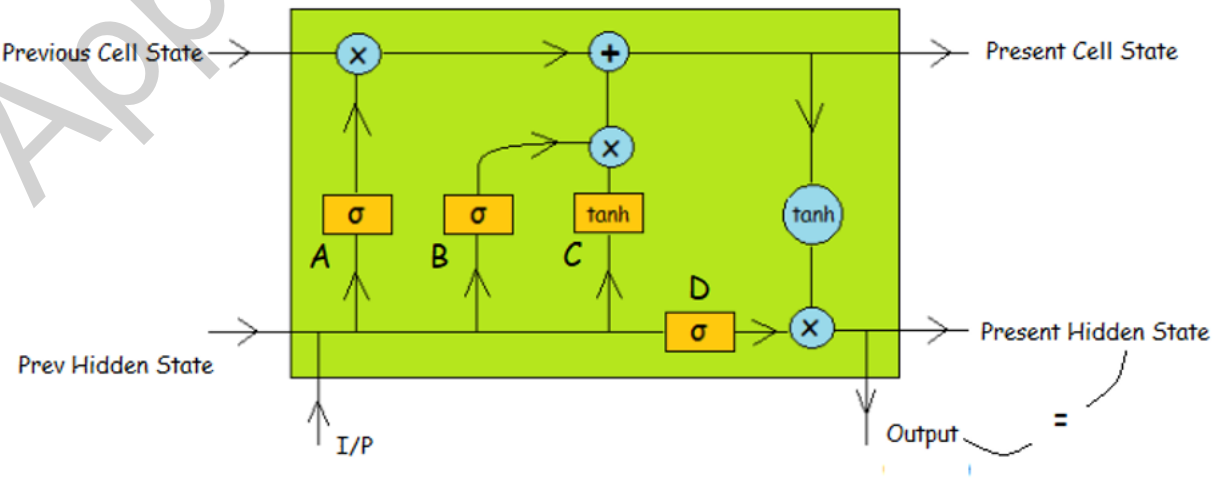
Thus the four neural layers in the LSTM can be considered as performing the following "learning":

- Layer A learns the weights that would produce the best possible importance vector with respect to the previous "memory" (ie: previous cell state vector).
- Layer C learns the weights that would produce the best possible information vector with respect to the current input. (ie: concatenation of W2 & W1').
- Layer B learns the weights that would produce the best possible importance vector with respect to Wc (output of layer C).
- Layer D learns the weights that would produce the best possible importance vector with respect to the current cell state vector passed through a tanh function (ie: Cell state normalized to be within the -1 to 1 range).

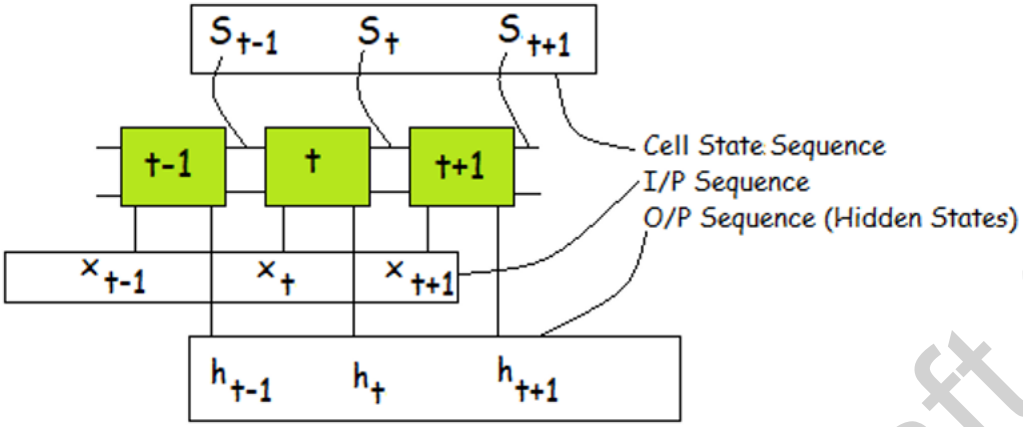
Note that while processing the first word vector W1 in the input sequence, the initial cell state vector S0 and the initial hidden state vector W0' are basically zero vectors.

LSTM REPRESENTATION:

The LSTM and all the operations that happen within it can be represented using the simplified diagrammatic representation shown below.



The image below represents a LSTM that is iterating through an input sequence represented by timesteps t-1, t & t+1.



KERAS LSTM LAYERS:

Shown below is the LSTM layer class in Keras, with the most relevant argument settings.

```
tf.keras.layers.LSTM(
    units,
    return_sequences=False,
    return_state=False,
    stateful=False)
```

The **units** argument lets us set the size of the output vectors of the output sequence.

The **return_sequence** keyword argument is used to determine whether the layer should return a sequence or a single output (in case last layer).

The **return_state** keyword argument if set to True will return the cell state vector as well as the output vector.

The **stateful** keyword argument if set to True will make the LSTM layer to 'remember' the cell state of the previous sequence/sentence that was processed. Thus while processing a sentence/sequence, the cell state derived from the previous sentence will be used as the current cell state input, instead of reinitializing it for every sentence/input sequence. This is normally set to False.

LSTM ON AMAZON REVIEWS DATASET:

For the sake of demonstration we will perform sentiment classification on the Amazon reviews dataset using a stacked LSTM model (ie:Many LSTM layers used in succession).

Unlike in the previous cases, where we used google word vector embeddings, in case of LSTMs we use embedding layers to learn unique embeddings for each word in the dataset. This requires the following steps:

1. Each word in the corpus is assigned/mapped to a specific integer greater than 1.
2. The words in each sentence in the dataset are then replaced by their respective integer mappings.

The two functions shown below performs the two steps mentioned above:

```
def fn_embed_words_as_nums(corpus):
    dict0_word_embeds = {}
    pbar = ProgressBar()
    for sentence in pbar(corpus):
        for word in sentence.split():
            if word not in dict0_word_embeds:
                # Assign a unique index to each unique word:
                dict0_word_embeds[word] = (len(dict0_word_embeds) + 1)
                # Note that we don't attribute index 0 to anything.

    return dict0_word_embeds
```

```
def fn_embed_docs(corpus, dict0_word_embeds, max_lg = 30):
    from keras.preprocessing.sequence import pad_sequences

    pbar = ProgressBar()
    list0_doc_vecs = []
    for doc in pbar(corpus):
        list0_words = doc.split()

        doc_vec = [dict0_word_embeds.get(word, 0) for word in list0_words]
        doc_vec = [num for num in doc_vec if num != 0][: max_lg]
        list0_doc_vecs.append(doc_vec)

    list0_padded_doc_vecs = pad_sequences(list0_doc_vecs)
    arry0_doc_vecs = np.array(list0_padded_doc_vecs)
    return arry0_doc_vecs
```

We perform step 1 as shown below:

```
corpus = df_nlp_tr.reviews.values
dict0_word_embeds = fn_embed_words_as_nums(corpus)
```

```
100% (147659 of 147659) |#################|
```

```
vocab_size = len(dict0_word_embeds)
vocab_size
```

```
65854
```

We then perform step 2 as shown below:

```
tr_corpus = df_nlp_tr.reviews.values
ts_corpus = df_nlp_ts.reviews.values

tr_embeddings = fn_embed_docs(tr_corpus, dict0_word_embeds, max_lg = 50)
ts_embeddings = fn_embed_docs(ts_corpus, dict0_word_embeds, max_lg = 50)

tr_embeddings.shape, ts_embeddings.shape
```

```
100% (147659 of 147659) |#################|
100% (36915 of 36915) |###################|
((147659, 50), (36915, 50))
```

Note that we restrict the length of the input sentence to the max size of 50 words. All sentences having less than 50 words are padded with zero values using the inbuilt keras **pad_sequences** function used within function **fn_embedded_docs**. Thus all sentences will now be represented by a 50 integer embedding.

Thus we have the final form of our train and test sets as shown below:

```
y_tr = df_nlp_tr.ratings.values
y_ts = df_nlp_ts.ratings.values

y_tr.shape, y_ts.shape
```

```
((147659,), (36915,))
```

```
df_tr = pd.DataFrame(tr_embeddings).assign(labels = y_tr)
df_ts = pd.DataFrame(ts_embeddings).assign(labels = y_ts)

df_tr.shape, df_ts.shape
```

```
((147659, 51), (36915, 51))
```

We the define our Keras LSTM model as shown below:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, LSTM, Dropout
```

```
model = Sequential(name = 'rnn_a')

model.add(Embedding(vocab_size+1, 100))

model.add(LSTM(100, return_sequences=True))
model.add(Dropout(0.1))
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(50, return_sequences=False))

model.add(Dense(1, activation='sigmoid'))

model.summary()
```

```
Model: "rnn_a"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 100)         6585500
_________________________________________________________________
lstm_9 (LSTM)                (None, None, 100)         80400
_________________________________________________________________
dropout_3 (Dropout)          (None, None, 100)         0
_________________________________________________________________
lstm_10 (LSTM)               (None, None, 100)         80400
_________________________________________________________________
lstm_11 (LSTM)               (None, 50)                30200
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 51
=================================================================
Total params: 6,776,551
Trainable params: 6,776,551
Non-trainable params: 0
_________________________________________________________________
```


Note the "Embedding" layer used in the LSTM model above. The embedding layer basically creates a randomly initialized unique vector for each of the integer encodings created for the words in the vocabulary. As can be seen the embedding layer needs to be specified 2 parameters:

1. The first parameter specifies the number of words in the vocabulary and thus the number of word vectors that need to be initialized. Thus we specify the value vocab_size + 1. This is because all the words are mapped to non zero integers. The zero integer is reserved to

represent padding. Thus the embedding layer learns unique word vector embeddings for each word in the vocabulary and for the padding in general.

2. The second parameter of the keras embedding layer is used to define the dimensionality of the word vector embeddings required. In the model defined above we use an embedding size of 100. Thus each of the words in the vocabulary will be internally represented by an unique 100 -D word vector.

Also note that except for the last LSTM layer, we set the **return_sequences** parameter to False. Thus we get a sequential output containing the same number of vectors from the first layer which form the sequential input to the next layer and so on. Only for the last layer **return_sequences** is set to False. Thus this layer will take in the sequence of vectors outputted by the previous layer and will output a single vector (ie: only the last vector in its output sequence) for every input sequence/sentence. This vector is then fed to a Conventional neural layer (one neuron in case of this model) and the output of this layer forms the final output of the model.

We train the model as shown below.

```
model = model
optimizer=tf.keras.optimizers.Adamax()

df_tr_, df_ts_ = df_tr, df_ts
model_save_path =  data_path + 'RNN_MODELS/model_a/'

batch_size, epochs = 150, 15
class_bias = {0:87, 1:13}

z = fn_NN_binary_clf(model, optimizer, df_tr_, df_ts_, model_save_path,
                     batch_size=batch_size, epochs=epochs,
                     class_weight = class_bias)

df_performance_model_rnn_2, df_history_rnn_2 = z
```

```
100% ████████████████████ 15/15
```

We then get a statistical overview of the performance of the models at different epochs as shown below:

```
regd_columns = 'tr_prec_1 tr_rec_1 tr_prec_0 tr_rec_0 diff_rec_1 diff_rec_0'.split()
df_performance_model_rnn_2.loc[:, regd_columns].describe()
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
count	15.000000	15.000000	15.000000	15.000000	15.000000	15.000000
mean	0.984756	0.849473	0.511704	0.922136	0.033688	0.125656
std	0.010158	0.048003	0.086707	0.047395	0.019906	0.075486
min	0.959778	0.729235	0.330004	0.813570	0.000488	0.002741
25%	0.979392	0.825256	0.456212	0.894090	0.023015	0.066479
50%	0.987048	0.857208	0.516727	0.931381	0.028927	0.139738
75%	0.992798	0.884281	0.576604	0.960882	0.048543	0.179482
max	0.995830	0.909256	0.638272	0.976774	0.064146	0.263945

We then filter out the best models as shown below:

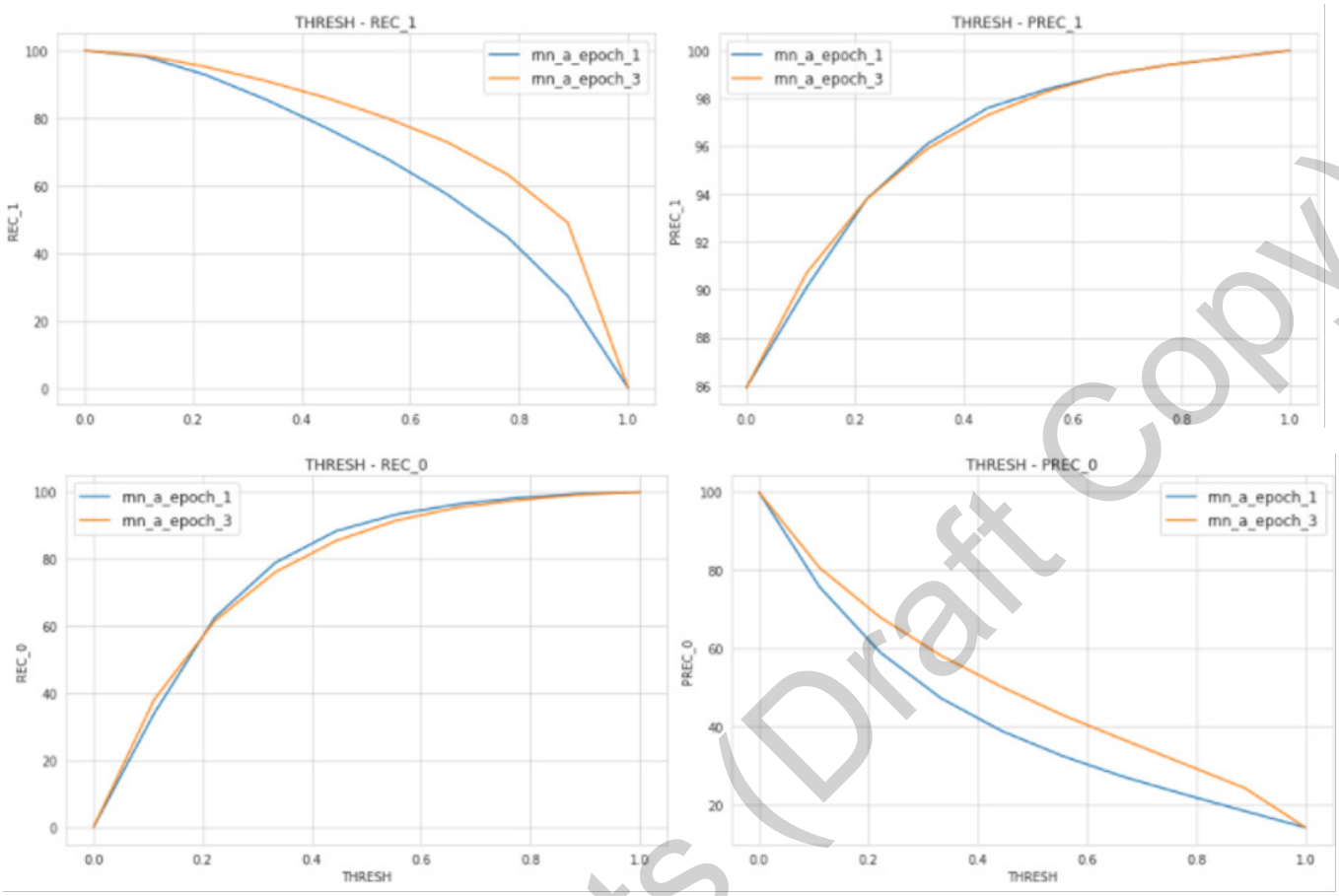
```
dff = df_performance_model_rnn_2
df1 = dff[dff.diff_rec_0 < 0.06]
df2 = df1[df1.diff_rec_1 < 0.06]
df3 = df2[df2.tr_rec_0 > 0.6]
df_filtered_2 = df3[df3.tr_rec_0 > 0.6]
df_filtered_2.loc[:, regd_columns]
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
rnn_a_epoch_1	0.971793	0.794881	0.407122	0.859252	0.028927	0.002741
rnn_a_epoch_3	0.978286	0.821422	0.449293	0.888777	0.022929	0.046884

We then evaluate the performance of the best models by checking their precision recall curves as shown below:

```
def fn_load(model_name, model_save_path = model_save_path):
    model_name = model_save_path + model_name + '.h5'
    return keras.models.load_model(model_name)
```

```
kwargs = dict(model_save_path = data_path + 'RNN_MODELS/model_a/')
list0_filtered_models_2 = [fn_load(i, **kwargs) for i in list(df_filtered_2.index)]
df_Xy_ = df_tr
legend = list(df_filtered_2.index)

fn_performance_models_data(list0_filtered_models_2, df_Xy_, legend, NN=True)
```



Assuming we want to optimize for recall of the negative class we choose the second model thresholded at 0.5 for our prediction purposes.

We then test out chosen model for generalization as shown below:

```
df_Xy_ = df_tr_

model_ = listO_filtered_models_2[-1]
thresh = 0.5

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.3453
ACCURACY: 83.979
--------------------
```

	prec	rec
class_0	46.4	88.9
class_1	97.9	83.2

```
df_Xy_ = df_ts_

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.374
ACCURACY: 81.983
```

	prec	rec
class_0	42.7	81.4
class_1	96.4	82.1

As can be seen from the outputs above, the chosen model gives > 80% recall for both classes and generalizes reasonably well to the test set.

BIDIRECTIONAL LSTM:

Bidirectional LSTM layers are a slight modification of the convention LSTM layer. A Bidirectional LSTM layer basically consists of two LSTM layers where one of them iterates through the input sequence in the reverse order (right to left). The outputs of these two layers are either concatenated/summed/multiplied/averaged to form the final output of the Bidirectional layer.

Since Bidirectional LSTMs interpret the input sequences using both forward and backward directions they generally tend to give a better performance than conventional LSTM layers.

KERAS BIDIRECTIONAL LSTM LAYER:

Shown below is the Keras Bidirectional LSTM class.

```
tf.keras.layers.Bidirectional(layer,
                              merge_mode="concat")
```

The **layer** argument shown above refers to the keras LSTM layer class (along with the required configuration) that will be used to create two internal LSTM layers.

The **merge_mode** keyword argument is used two define how the outputs of the two internal LSTMs are going to be combined. The default mode is "concat". Other options include: 'sum', 'mul', & 'ave' (sum, multiply and average)

BIDIRECTIONAL LSTM ON AMAZON DATASET:

Show below are the sequence of operations followed for training and evaluating the Bidirectional LSTM model.

DEFINING THE MODEL:

```
from tensorflow.keras.layers import  LSTM
```

```
model_b = Sequential(name = 'rnn_b')

model_b.add(Embedding(vocab_size+1, 100))

model_b.add(Bidirectional(LSTM(100, return_sequences=True)))
model_b.add(keras.layers.Dropout(0.1))

model_b.add(LSTM(100, return_sequences=True))
model_b.add(LSTM(50, return_sequences=False))

model_b.add(Dense(1, activation='sigmoid'))

model_b.summary()
```

```
Model: "rnn_b"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 100)         6585500
_________________________________________________________________
bidirectional (Bidirectional (None, None, 200)         160800
_________________________________________________________________
dropout (Dropout)            (None, None, 200)         0
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 100)         120400
_________________________________________________________________
lstm_2 (LSTM)                (None, 50)                30200
_________________________________________________________________
dense (Dense)                (None, 1)                 51
=================================================================
Total params: 6,896,951
Trainable params: 6,896,951
Non-trainable params: 0
_________________________________________________________________
```

MODEL TRAINING:

```
model = model_b
optimizer=tf.keras.optimizers.Adamax()

df_tr_, df_ts_ = df_tr, df_ts
model_save_path =  data_path + 'RNN_MODELS/model_b/'

batch_size, epochs = 150, 5
class_bias = {0:87, 1:13}

z = fn_NN_binary_clf(model, optimizer, df_tr_, df_ts_, model_save_path,
                     batch_size=batch_size, epochs=epochs,
                     class_weight = class_bias)

df_performance_model_rnn_b, df_history_rnn_b = z
```

```
100% 5/5
```

FILTERING OUT BEST MODELS:

```
regd_columns = 'tr_prec_1 tr_rec_1 tr_prec_0 tr_rec_0 diff_rec_1 diff_rec_0'.split()
```

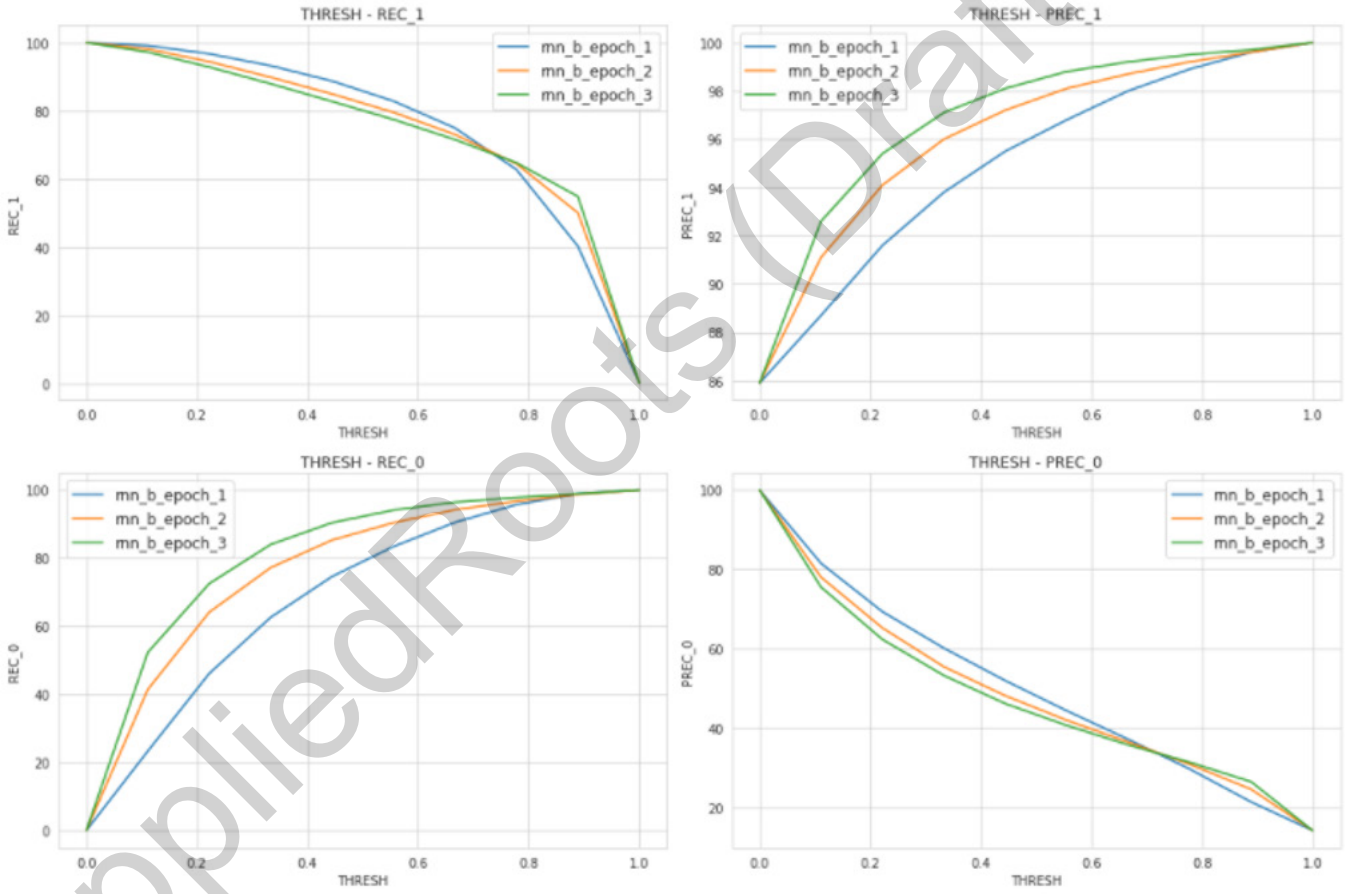
```
dff = df_performance_model_rnn_b
df1 = dff[dff.diff_rec_0 < 0.06]
df2 = df1[df1.diff_rec_1 < 0.06]
df3 = df2[df2.tr_rec_0 > 0.6]
df_filtered_b = df3[df3.tr_rec_0 > 0.6]
df_filtered_b.loc[:, regd_columns]
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
rnn_b_epoch_1	0.972631	0.798917	0.412942	0.862858	0.012976	0.038277
rnn_b_epoch_2	0.976975	0.809385	0.431787	0.883631	0.023662	0.032699
rnn_b_epoch_3	0.979522	0.821973	0.451834	0.895172	0.039308	0.040777

MODEL EVALUATION:

```
def fn_load(model_name, model_save_path = model_save_path):
    model_name = model_save_path + model_name + '.h5'
    return keras.models.load_model(model_name)
```

```
kwargs = dict(model_save_path = data_path + 'RNN_MODELS/model_b/')
list0_filtered_models_b = [fn_load(i, **kwargs) for i in list(df_filtered_b.index)]
df_Xy_ = df_tr
legend = list(df_filtered_b.index)

fn_performance_models_data(list0_filtered_models_b, df_Xy_, legend, NN=True)
```



Assuming we want to optimize for recall of the negative class we choose the second model thresholded at 0.5 for our prediction purposes.

MODEL TESTING:

We then test out chosen model for generalization as shown below:

```
df_Xy_ = df_tr

model_ =  list0_filtered_models_b[-2]
thresh = 0.5

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.3614
ACCURACY: 83.062
--------------------
```

	prec	rec
class_0	44.8	88.1
class_1	97.7	82.2

```
df_Xy_ = df_ts

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.388
ACCURACY: 81.368
--------------------
```

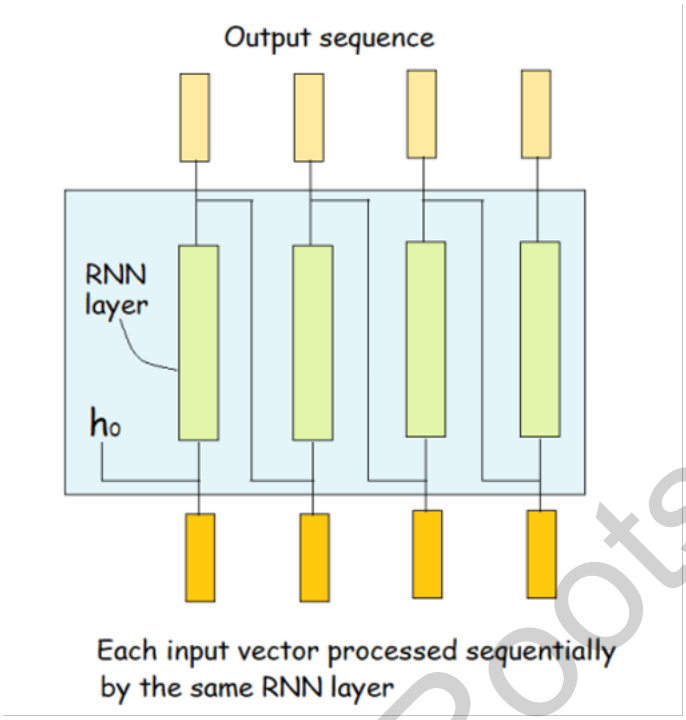
	prec	rec
class_0	41.8	82.5
class_1	96.6	81.2

As can be seen from the outputs above, the chosen model gives > 80% recall for both classes and generalizes reasonably well to the test set.

07. TRANSFORMERS

TRANSFORMERS

In the previous chapter on LSTMs, we saw that the LSTM is capable of Sequence to sequence tasks, where given a sequence of input vectors it produces an output sequence of the same size. The image below expresses this concept schematically.

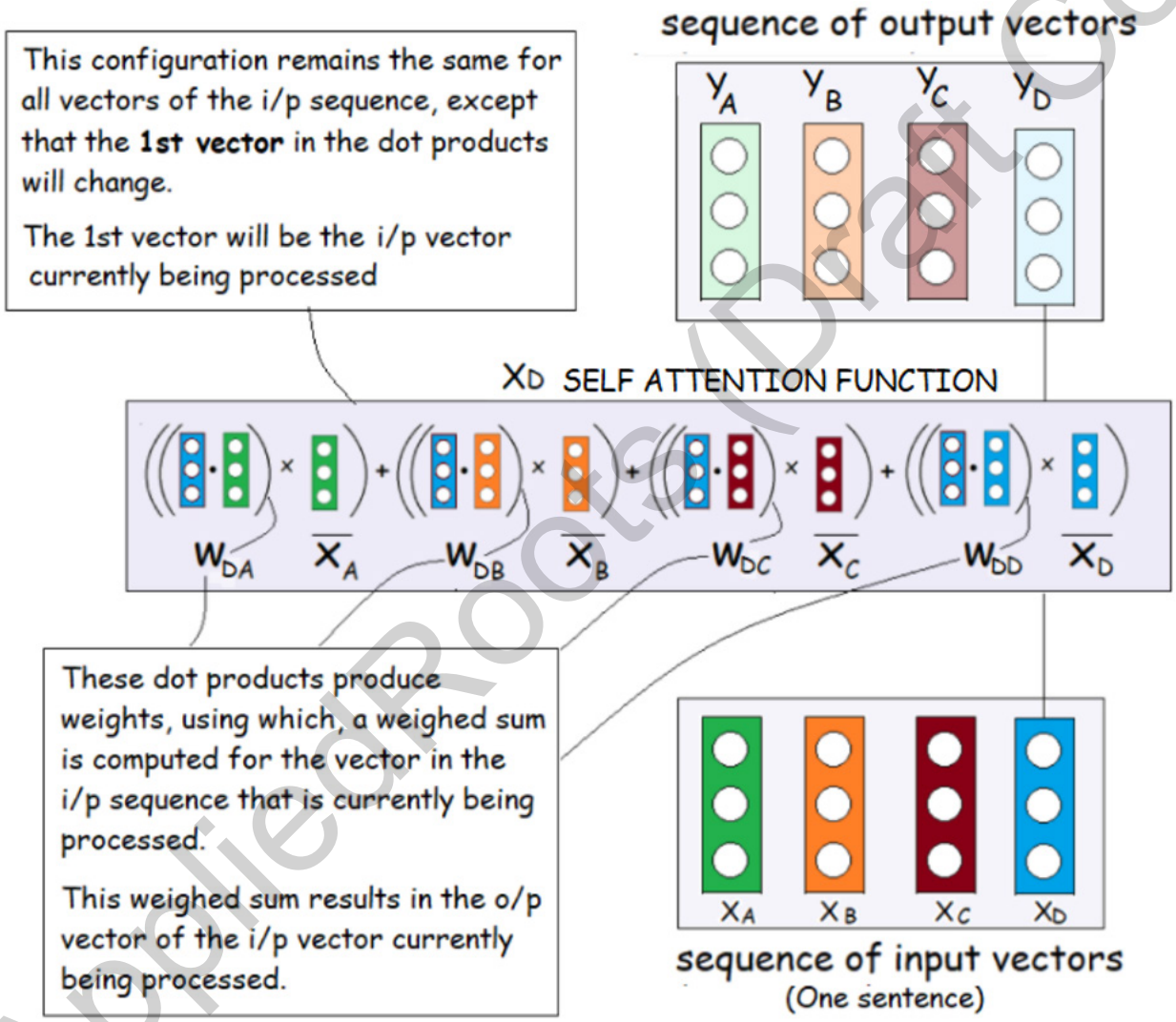


LSTMs incorporated sequential information into its model and made predictions based on it. Though the sequence of the words in a sentence holds meaning, it is not all there is. In reality, language is non linear and every word in a sentence has unique relations with all the other words it contains. We generally read an entire sentence, understand the correlations that exist between the words it contains and then pay more attention to certain words that seem to express essential information by the relationship they have with each other.

Consider the sentence “The story had just begun”. Here the key words are “story” and “begun” since they, in relation to each other, contain most information. Hence when we examine one word we pay attention to the other and vice versa. Transformers are a neural network architecture that incorporates the concept of Attention into its design. It does so by making use of the Self Attention Function at its core.

THE SELF ATTENTION FUNCTION:

Self attention is a sequence to sequence function, where the inputs and outputs are sequences of vectors. The image shown below schematically describes the self attention function. Each vector in the input sequence has a vector corresponding to it in the output sequence. This corresponding output vector is the outcome of a weighted sum that involves all other vectors in the input sequence. The image below describes how the output vector ($\mathbf{y}_D$) corresponding to the last vector in the input sequence ($\mathbf{X}_D$) is generated.



Thus, the output of XD can be expressed as:

$$\mathbf{y_D = W_{DA}\,(X_A) + W_{DB}\,(X_B) + W_{DC}\,(X_C) + W_{DD}\,(X_D)}$$

Where all the weights **W** are scalars resulting from dot products.

In general $\mathbf{yi = \Sigma W_{i,j}(X_j)}$ for every word (**j**) in the input sequence.

The expression above is not completely correct. There is one more step that needs to be performed before the self attention function is completely defined. This step is: All the weights used in the above definition of $\mathbf{y}_i$ have to be grouped together in the same order as the input sequence, to form a vector and then passed through a **softmax** function. Thus all weights will now add up to one. These weights are then used within the attention function described above.

Given an vector from the input sequence, what is happening in the Self Attention Function is this:

1. Each element in the input sequence is scaled by a weight and summed to produce the corresponding vector in the output sequence.
2. Each weight corresponding to a particular input vector, represents the similarity of that input vector with respect to the vector currently being processed Xcurrent . X_i

THE ATTENTION FUNCTION:

The Self Attention function described above is basically a type of attention function. Basic attention functions are sequence to sequence functions, which take as input, two same sized vector sequences (say: X1 and X2) that result in one output vector sequence Y of the same size. So using the same notation as in the above image, the two input sequences can be denoted as:

$X1 = [X_A^{(1)}, X_B^{(1)}, X_C^{(1)}, X_D^{(1)}]$ & $X2 = [X_A^{(2)}, X_B^{(2)}, X_C^{(2)}, X_D^{(2)}]$

The output vector corresponding to XD(1) (ie: yD) is then computed using weights derived from X2 as shown below:

$$\mathbf{y_D = W_{DA}\,(X_A^{(1)}) + W_{DB}\,(X_B^{(1)}) + W_{DC}\,(X_C^{(1)}) + W_{DD}(X_D^{(1)})}$$

Where:

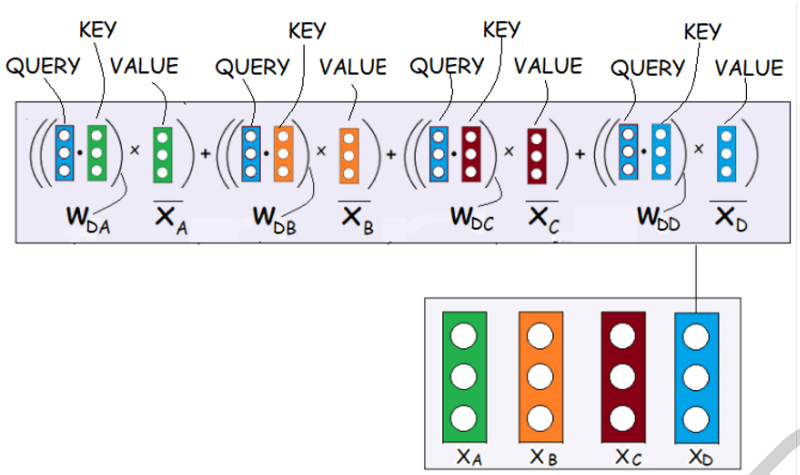
$$\mathbf{W_{DA}\,,\, W_{DB}\,,\, W_{DC}\,,\, W_{DD}} =$$

$$\mathbf{softmax(}(X_D^{(1)}.\, X_A^{(2)}),\ (X_D^{(1)}.\, X_B^{(2)}),\ (X_D^{(1)}.\, X_C^{(2)}),\ (X_D^{(1)}.\, X_D^{(2)}))$$

The attention function can be used to compare two sentences or phrases and figure out whether they are referring to the same topic or contain the same information/meaning. But we can just as easily use it to compare a sentence with itself. This results in Self Attention.

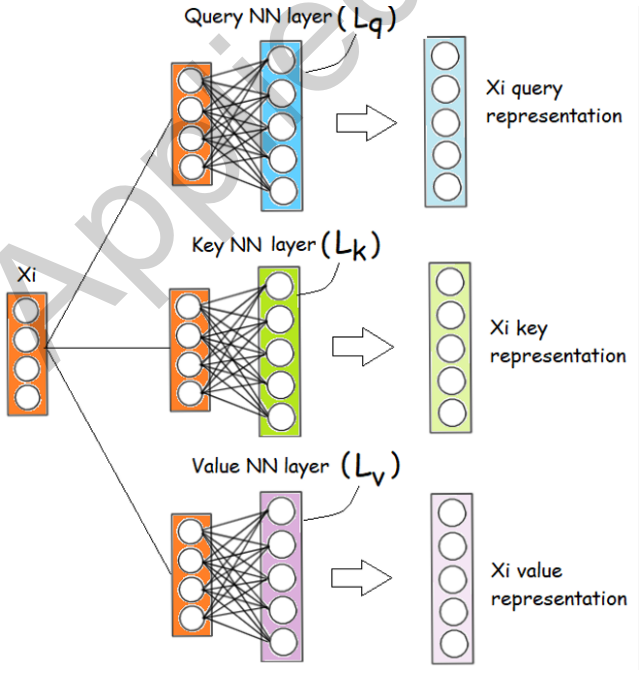
MAKING THE SELF ATTENTION FUNCTION TRAINABLE:

When we observe the self attention function, we see that every vector in the input sequence is used in three different ways: As a query vector, as a key vector and as a value vector. The meaning of these three roles is expressed in the image below.



To make the Self Attention function trainable we introduce 3 neural network layers Lq, Lk and Lv that have no activation functions (ie: they perform linear transfor - mations). Every vector in the input sequence is passed through each of these neural network layers to get their query, key and value representations. Once we have all query, key and value representations of each of the input vectors, we perform the self attention computation described earlier using these representative vectors instead of the actual vectors in the input sequence.

The image below describes how each input vector is passed through three neural network layers to get their query, key and value vector representations.



Thus every input vector Xi will have three vectors associated with it, which are linear transformations obtained by passing them through the three neural network layers described above. They are:

1. Its query representation: Xi query
2. Its key representation: Xi key
3. Its value representation: Xi value

SCALING THE DOT PRODUCT BETWEEN THE QUERY AND KEY VECTORS:

As discussed earlier, the weights computed from the dot product of the query and key vectors are first collected to form a vector and then this vector is passed through a softmax function as shown below (wrt the image describing self attention function shown previously):

$$[W_{DA}, W_{DB}, W_{DC}, W_{DD}] = softmax([W'_{DA}, W'_{DB}, W'_{DC}, W'_{DD}])$$

Where:

W'_{DA} = XD_query . XA_key

W'_{DB} = XD_query . XB_key

W'_{DC} = XD_query . XC_key

W'_{DD} = XD_query . XC_key

The values of the resulting vector (ie: $[W_{DA}, W_{DB}, W_{DC}, W_{DD}]$) are then used to compute the weighted sum of the input vectors in the sequence.

Now, the softmax function is sensitive to very large input values. So if the dot products described above produce very large values, then the resulting high variance in the softmax output can create unsteady gradients and impair the learning process.

The average value of a dot product grows with the dimensionality of the vectors between which the dot product is taken. Say we have a unit vector with **d** dimensions, the value of its magnitude will be $\sqrt{d}$ (we are dividing out the amount by which the increase in dimension increases the magnitude of the vectors). Thus we scale the value of each of the inputs to the softmax function (ie:W'_{DA}, W'_{DB}, W'_{DC} **and** W'_{DD}) by dividing them by $\sqrt{d}$. This stops the inputs to the softmax function from growing too large.

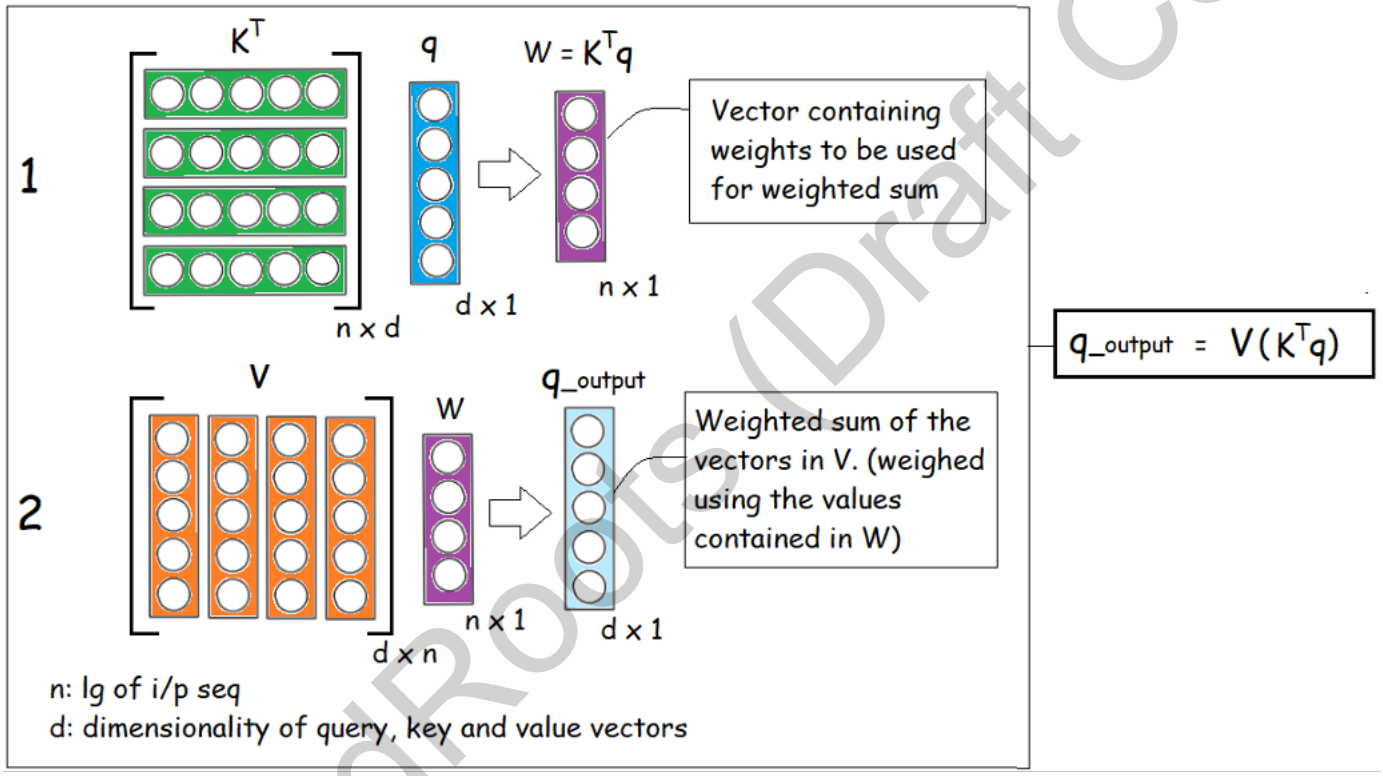
MATRIX REPRESENTATION OF SELF ATTENTION FUNCTION:

Now if we stack all the key and value vectors of an input sequence in separate matrices, we will get a key matrix(**K**) and a value matrix(**V**) each of size **d x n**, where **n** is the number of vectors in the input sequence and **d** as mentioned earlier, is the dimensionality of the query, key and value vectors. Then using these matrices we can schematically represent the Self Attention computation for a given query, say X_{D_query} (ie: query vector of last word in the input

sequence, with respect to the example discussed previously) as shown in the image below.

STEP 1:

The query vector **q** is transformed by the key matrix $\mathbf{K^T}$. This is the same as computing the dot product between the query vector **q** and each of the key vectors corresponding to the **n** elements in the input sequence. The result of this matrix transformation ($\mathbf{K^Tq}$) is a **n** dimensional vector **W**. Each value in **W** represents the weights with which each of the n "**value**" vectors will be multiplied (or scaled) with. This happens in the second step.



STEP 2:

We then perform matrix multiplication between matrix V and vector W as shown in the image above. This is equivalent to scaling each vector in V by the corresponding weights in W and adding them (ie: weighted sum). We thus get the output vector corresponding to the query vector inputted (ie: **q_output**).

Now instead of considering just one query vector, let us consider all the query vectors of the input sequence. This will only require changing the input from a single query vector to a matrix containing all the query vectors corresponding to each vector in the input sequence.

Thus the equation becomes:

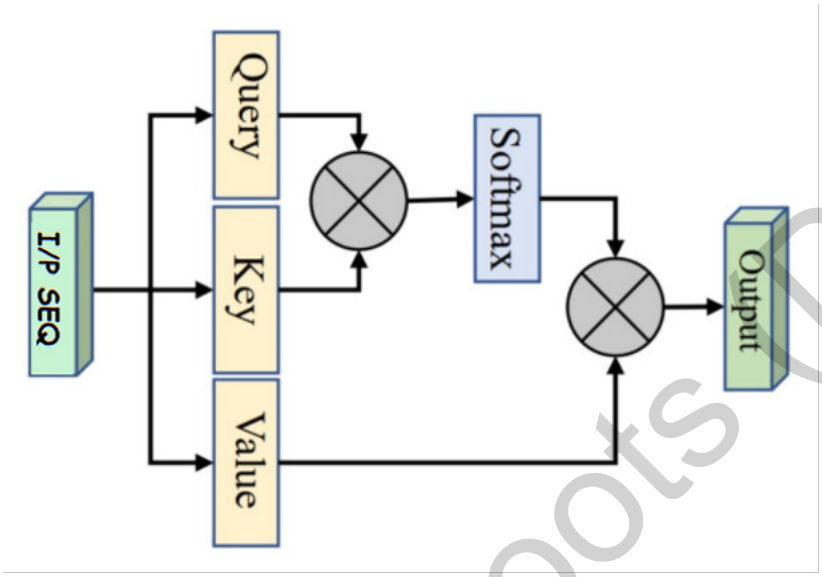
$\mathbf{Q\_output = V(K^TQ)}$

Where Q is a d x n matrix containing n query vectors of d dimensions each and Q_output is also a d x n matrix containing n output vectors of d dimensions each.

The above equation is further modified to represent the attention function accurately as shown below.

$$\text{Q\_output} = V \,.\, \text{softmax}\left((K^T Q) \,/\, \sqrt{d} \right)$$

The image below provides a simplified perspective of the attention mechanism.

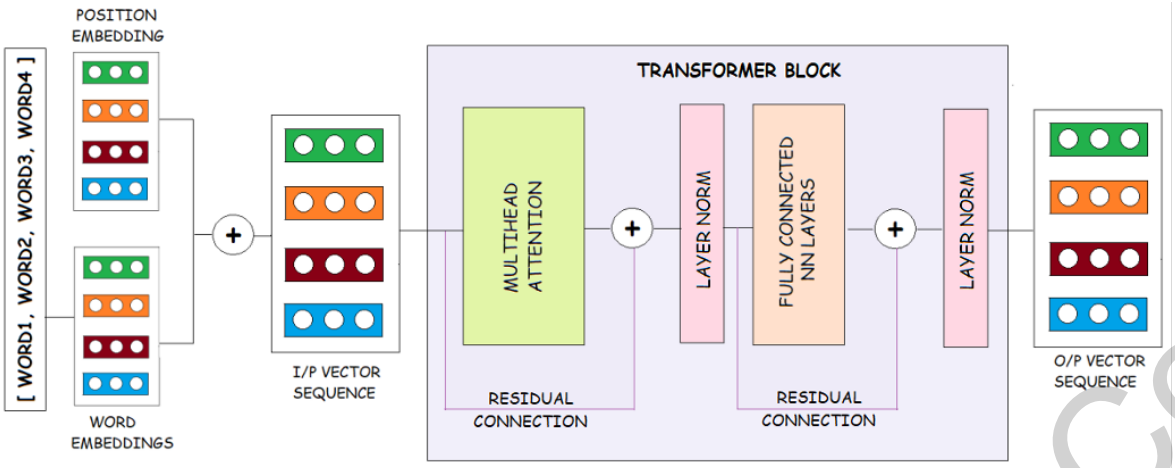


MULTI HEAD SELF ATTENTION:

Multi head attention refers to the usage of multiple attention functions (or attention heads) over the same input sequences. This is similar to the usage of multiple kernels in a CNN layer. Just like each kernel in a CNN layer learns a different set of features from the images, here also each attention head learns different attention relationships within the input sequences. The outputs of these attention functions are then concatenated and a linear transformation applied over them so as to reduce the dimensions of the output vectors back to **d**.

THE TRANSFORMER BLOCK:

A transformer is not just about self attention layers, it is an architecture, which advantageously uses residual connections and layer normalization. In general a transformer "block" has a schematic as shown in the image below.



POSITIONAL ENCODING:

Unlike RNNs which see the input fed to it as a **sequence** of vectors, Self Attention sees its input as a **set**. It does not take into account the sequential order of the input vectors. If we permute the input sequence, the output sequence will be exactly the same, except permuted the same way as the input.

To incorporate positional/sequence information into the transformer model, we create encodings for each position in the input sequence. These encodings will have the same dimensionality as that of the vectors in the input sequence. The positional encoding vectors are then added with their corresponding input vectors to create the final input vector sequence.

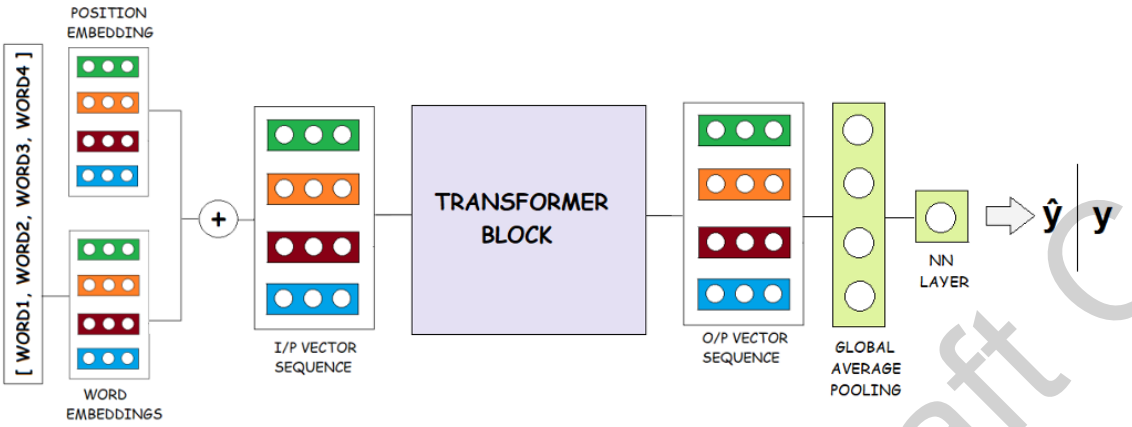
In the previous chapter on LSTMs, we encoded each word in the input data corpus with a unique integer and then used a keras embedding layers of the desired embedding size to "learn" word embeddings for each input integer (ie: word), during training.

We do the same here, but apart from using an embedding layer for creating word embeddings, we also use another embedding layer of the same size to create embedding for each position within a sequence. So given a maximum sequence length of say hundred, we create an embedding layer that has a "vocabulary" size of hundred and dimensionality the same as that of the input embedding. During training the weights for these positional embeddings are also learnt.

TRANSFORMER ARCHITECTURE FOR CLASSIFICATION:

The Transformer block can be adapted for binary classification purposes by using the arrangement shown in the image below. The output from the Transformer Block is passed through a global average pooling layer, which takes the average value of each of the vectors in the output sequence and outputs the resulting vector. This resulting vector can thenv

through a series of neural network layers and reduced in size to eventually output a single value, which is passed through a sigmoid activation, to give a value between 0 and 1. One can then use log loss to compute the loss between the predicted value and labels.



SENTIMENT CLASSIFICATION USING TRANSFORMER:

For the sake of demonstration we will use the amazon reviews dataset for sentiment classification. Just as in the case of LSTMs, we encode the input corpus with an unique integer for each word as shown below.

```
data_path = 'gdrive/My Drive/3A_PGD/CODE_PGD/TRANSFORMERS/'

df_nlp_tr = pd.read_csv(data_path + 'df_nlp_tr.csv')
df_nlp_ts = pd.read_csv(data_path + 'df_nlp_ts.csv')

df_nlp_tr.sample(5)
```

	prod_id	reviews	ratings
39847	b008zrkzsm	tastes exactly like regular fatty creamy peanu...	1
34732	b000g17682	this soup was a monumental disappointment as n...	0
17862	b004ijmvqk	perfect healthy world conscious snack the cook...	1
35937	b0041cir62	the back of the package gives recipes that are...	0
97925	b00014d37w	i ve started cooking with coconut oil and have...	1

```
def fn_embed_words_as_nums(corpus):
    dict0_word_embeds = {}
    pbar = ProgressBar()
    for sentence in pbar(corpus):
        for word in sentence.split():
            if word not in dict0_word_embeds:
                # Assign a unique index to each unique word:
                dict0_word_embeds[word] = (len(dict0_word_embeds) + 1)
                # Note that we don't attribute index 0 to anything.

    return dict0_word_embeds
```

```
def fn_embed_docs(corpus, dict0_word_embeds, max_lg = 100):
    from keras.preprocessing.sequence import pad_sequences

    pbar = ProgressBar()
    list0_doc_vecs = []
    for doc in pbar(corpus):
        list0_words = doc.split()

        doc_vec = [dict0_word_embeds.get(word, 0) for word in list0_words]
        doc_vec = [num for num in doc_vec if num != 0][: max_lg]
        list0_doc_vecs.append(doc_vec)

    list0_padded_doc_vecs = pad_sequences(list0_doc_vecs)
    arry0_doc_vecs = np.array(list0_padded_doc_vecs)
    return arry0_doc_vecs
```

```
corpus = df_nlp_tr.reviews.values
dict0_word_embeds = fn_embed_words_as_nums(corpus)
```

```
100% (147659 of 147659) |################| Elapsed Time: 0:00:02 Time:  0:00:02
```

```
vocab_size = len(dict0_word_embeds) + 1
vocab_size
```

```
65855
```

```
tr_corpus = df_nlp_tr.reviews.values
ts_corpus = df_nlp_ts.reviews.values

tr_embeddings = fn_embed_docs(tr_corpus, dict0_word_embeds, max_lg = 100)
ts_embeddings = fn_embed_docs(ts_corpus, dict0_word_embeds, max_lg = 100)

tr_embeddings.shape, ts_embeddings.shape
```

```
100% (147659 of 147659) |################| Elapsed Time: 0:00:04 Time:  0:00:04
100% (36915 of 36915) |##################| Elapsed Time: 0:00:01 Time:  0:00:01
((147659, 100), (36915, 100))
```

```
tr_embeddings
```

```
array([[   1,    2,    3, ...,   69,   70,   71],
       [  45,   47,   83, ...,   45,  136,   26],
       [ 176,   72,   10, ...,  246,   10,  247],
       ...,
       [   0,    0,    0, ...,  161,   20, 1060],
       [   0,    0,    0, ...,  548,   26,  151],
       [  52, 1504,   47, ...,   52, 2945, 2211]], dtype=int32)
```

```
y_tr = df_nlp_tr.ratings.values
y_ts = df_nlp_ts.ratings.values

y_tr.shape, y_ts.shape
```

```
((147659,), (36915,))
```

```
df_tr = pd.DataFrame(tr_embeddings).assign(labels = y_tr)
df_ts = pd.DataFrame(ts_embeddings).assign(labels = y_ts)

df_tr.shape, df_ts.shape
```

```
((147659, 101), (36915, 101))
```

We then implement our transformer model for sentiment classification as shown below using keras' functional API.

```
def fn_TRANSFORMER_BLOCK(vocab_size = vocab_size, max_sentence_lg = 100,
                         embed_dim = 32, num_heads = 5, ff_dim = 32):

    input = layers.Input(shape = (max_sentence_lg,))
    positions = np.array(range(max_sentence_lg))

    pos_emb = layers.Embedding(input_dim=max_sentence_lg, output_dim=embed_dim)(positions)
    word_emb = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)(input)
    emb = word_emb + pos_emb

    attn_output = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)(emb, emb)
    attn_output = layers.Dropout(0.3)(attn_output)
    out = layers.LayerNormalization()(emb + attn_output)
    x = layers.Dense(ff_dim, activation="relu")(out)
    x = layers.Dense(embed_dim)(x)
    x = layers.Dropout(0.1)(x)
    output = layers.LayerNormalization()(out + x)

    output = layers.GlobalAveragePooling1D()(output)
    output = layers.Dropout(0.1)(output)
    output = layers.Dense(20, activation="relu")(output)
    output = layers.Dropout(0.1)(output)
    output = layers.Dense(20, activation="relu")(output)
    output = layers.Dropout(0.1)(output)
    output = layers.Dense(1, activation="sigmoid")(output)

    model = Model(input, output)
    return model
```

A

B

C

CODE A: This code block implements data embedding using keras' embedding layers. Note that for every input, an integer encoding of each position is created and just like in the case of word embedding, for every integer representing position, a 32 dimensional embedding is created. Thus we end up with 2 equi sized sequences of vectors - one representing embedded words and the other, embedded positions. These are then added and fed to the MultiHeadAttention layer.

CODE B: This block implements the internals of the Transformer Block as shown in the image under the previous "Transformer block" subheading. We use the Keras' MultiheadAttention layer to do this. The main arguments and keyword entries for this MultiheadAttention layer is shown in the image below.

```
tf.keras.layers.MultiHeadAttention(
    num_heads,
    key_dim,
    value_dim=None,
    kernel_initializer="glorot_uniform")
```

When value dimensions are set to None, the same dimensions as those given for key_dim are used. Note that in the fn_TRANSFORMER_BLOCK code, we give two inputs to the MultiHeadAttention layer and both are the same Embedded input sequence of vectors. Thus we get "self attention". We use five attention heads in this function.

CODE C: This code uses keras' GlobalAveragePooling1D layer to perform average pooling of each vector in the sequence outputted by the Transformer Block. The resulting vector is then passed through feed forward dense layers and the dimensionality of the vector eventually reduced to one. The activation function of the last layer is set to "sigmoid".

We then compile our model and check its summary as shown below.

```
transformer = fn_TRANSFORMER_BLOCK()
transformer.summary()
```

```
Model: "model"
```

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_3 (InputLayer)            [(None, 100)]        0
__________________________________________________________________________________________________
embedding_5 (Embedding)         (None, 100, 32)      2107360     input_3[0][0]
__________________________________________________________________________________________________
tf.__operators__.add_6 (TFOpLam (None, 100, 32)      0           embedding_5[0][0]
__________________________________________________________________________________________________
multi_head_attention_2 (MultiHe (None, 100, 32)      20992       tf.__operators__.add_6[0][0]
                                                                 tf.__operators__.add_6[0][0]
__________________________________________________________________________________________________
dropout_8 (Dropout)             (None, 100, 32)      0           multi_head_attention_2[0][0]
__________________________________________________________________________________________________
tf.__operators__.add_7 (TFOpLam (None, 100, 32)      0           tf.__operators__.add_6[0][0]
                                                                 dropout_8[0][0]
__________________________________________________________________________________________________
layer_normalization_4 (LayerNor (None, 100, 32)      64          tf.__operators__.add_7[0][0]
__________________________________________________________________________________________________
dense_8 (Dense)                 (None, 100, 32)      1056        layer_normalization_4[0][0]
__________________________________________________________________________________________________
dense_9 (Dense)                 (None, 100, 32)      1056        dense_8[0][0]
__________________________________________________________________________________________________
dropout_9 (Dropout)             (None, 100, 32)      0           dense_9[0][0]
__________________________________________________________________________________________________
tf.__operators__.add_8 (TFOpLam (None, 100, 32)      0           layer_normalization_4[0][0]
                                                                 dropout_9[0][0]
__________________________________________________________________________________________________
layer_normalization_5 (LayerNor (None, 100, 32)      64          tf.__operators__.add_8[0][0]
__________________________________________________________________________________________________
global_average_pooling1d_2 (Glo (None, 32)           0           layer_normalization_5[0][0]
__________________________________________________________________________________________________
dropout_10 (Dropout)            (None, 32)           0           global_average_pooling1d_2[0][0]
__________________________________________________________________________________________________
dense_10 (Dense)                (None, 20)           660         dropout_10[0][0]
__________________________________________________________________________________________________
dropout_11 (Dropout)            (None, 20)           0           dense_10[0][0]
__________________________________________________________________________________________________
dense_11 (Dense)                (None, 20)           420         dropout_11[0][0]
__________________________________________________________________________________________________
dropout_12 (Dropout)            (None, 20)           0           dense_11[0][0]
__________________________________________________________________________________________________
dense_12 (Dense)                (None, 1)            21          dropout_12[0][0]
==================================================================================================
Total params: 2,131,693
Trainable params: 2,131,693
Non-trainable params: 0
```

We then train the model as shown below:

```
model = transformer
optimizer=tf.keras.optimizers.Adamax()

df_tr_, df_ts_ = df_tr, df_ts
model_save_path =  data_path + 'TRANSFORMER_MODELS/model_A/'

batch_size, epochs = 250, 15
class_bias = {0:85, 1:15}

z = fn_NN_binary_clf(model, optimizer, df_tr_, df_ts_, model_save_path,
                     batch_size=batch_size, epochs=epochs,
                     class_weight = class_bias)

df_performance_model_2, df_history_2 = z
```

```
100% ████████████████████ 15/15
```

We get an overview of the training results as shown below:

```
regd_columns = 'tr_prec_1 tr_rec_1 tr_prec_0 tr_rec_0 diff_rec_1 diff_rec_0'.split()
df_performance_model_2.loc[:, regd_columns].describe()
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
count	15.000000	15.000000	15.000000	15.000000	15.000000	15.000000
mean	0.969475	0.833751	0.455348	0.840762	0.003285	0.009922
std	0.012229	0.023705	0.049354	0.059821	0.005427	0.024081
min	0.929934	0.765665	0.311234	0.647328	0.000668	0.000298
25%	0.969358	0.827495	0.443905	0.840364	0.001124	0.001068
50%	0.973872	0.840561	0.469918	0.862403	0.001261	0.001886
75%	0.976075	0.848984	0.486567	0.873050	0.002323	0.004851
max	0.977284	0.856712	0.501311	0.878544	0.021829	0.094638

We then filter for the best models as shown below:

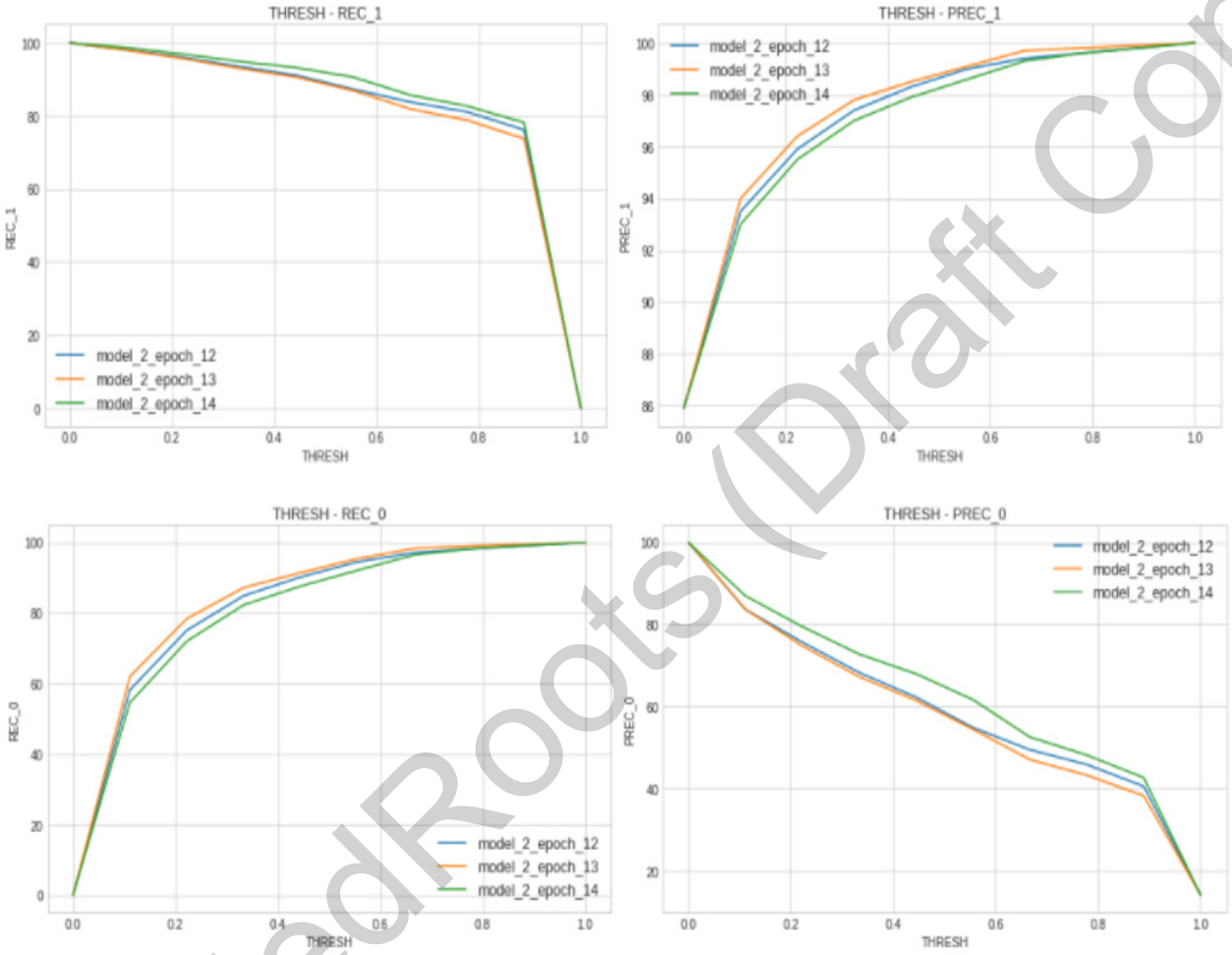
```
dff = df_performance_model_2
df1 = dff[dff.diff_rec_0 < 0.03]
df2 = df1[df1.diff_rec_1 < 0.03]
df3 = df2[df2.tr_rec_0 > 0.85]
df_filtered_2 = df3[df3.tr_rec_1 > 0.85]
df_filtered_2.loc[:, regd_columns]
```

	tr_prec_1	tr_rec_1	tr_prec_0	tr_rec_0	diff_rec_1	diff_rec_0
model_2_epoch_12	0.976712	0.852202	0.492882	0.876075	0.001005	0.000754
model_2_epoch_13	0.977072	0.854299	0.496864	0.877712	0.001163	0.000298
model_2_epoch_14	0.977284	0.856712	0.501311	0.878544	0.000668	0.000843

We check the precision recall performances of each model as shown below:

```
def fn_load(model_name, model_save_path = model_save_path):
    model_name = model_save_path + model_name + '.h5'
    return keras.models.load_model(model_name)
```

```
kwargs = dict(model_save_path = data_path + 'TRANSFORMER_MODELS/model_A/')
list0_filtered_models_2 = [fn_load(i, **kwargs) for i in list(df_filtered_2.index)]
df_Xy_ = df_tr
legend = list(df_filtered_2.index)

fn_performance_models_data(list0_filtered_models_2, df_Xy_, legend, NN=True)
```



Assuming we are optimizing for recall of the negative class, we choose the model at epoch_14, thresholded at 0.65 for our prediction purposes.

We further check the performance of our model over the train and test sets as shown below:

```
df_Xy_ = df_tr_

model_ =  list0_filtered_models_2[-1]
thresh = 0.65

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.1903
ACCURACY: 87.68
--------------------
```

	prec	rec
class_0	53.5	96.2
class_1	99.3	86.3

```
1   df_Xy_ = df_ts_
2
3   fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.2946
ACCURACY: 84.053
--------------------
```

	prec	rec
class_0	46.3	82.9
class_1	96.8	84.2

Thus we achieve a recall of around 83% for the negative class and around 84% for the positive class on the test set.