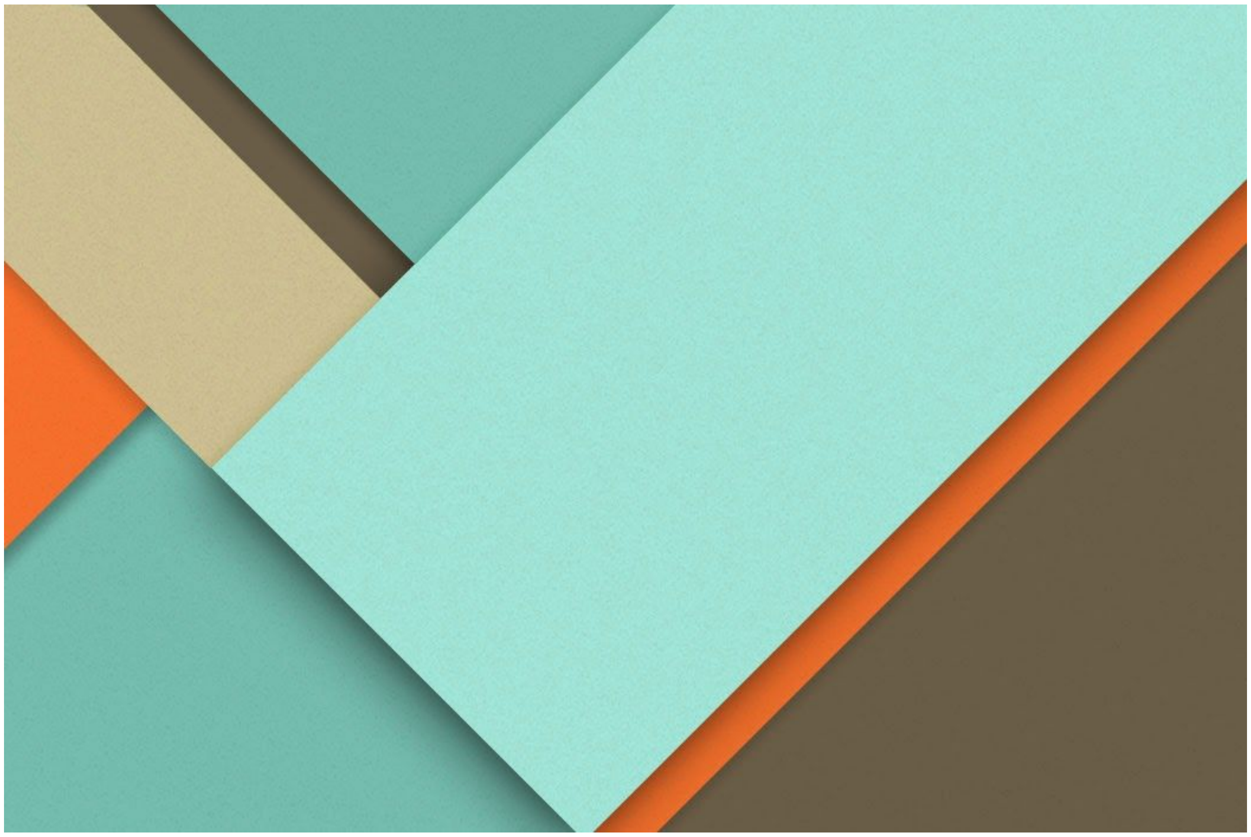


DIS Final Report - Deep Learning Applications

By Yousef Al-Kafif

Znumber : 23373185

TABLE OF CONTENTS:-

Overview

This directed independent study course's major intention is to study the fundamentals of Deep Learning and its applications to various scientific and engineering problems.

Overall Outcome

1. To be able to implement Deep Learning techniques and develop research skills.

Method of Learning

There are plenty of resources online for Deep Learning, but I preferred to use an online course format rather than learning on the go via bits of information through videos and blogs. The 2 online courses I utilized mainly are:

I. Deep Learning A-Z™: Hands-On Artificial Neural Networks

This is an online course offered on Udemy by 2 established data scientists. It focused on the intuition and concepts behind Artificial Neural Networks, Convolutional Neural Networks, Recurrent Neural Networks, Self Organizing Maps, and briefly over AutoEncoders. It also walked through the common coding practices of Deep Learning on Python utilizing the Keras library which ran on the Tensorflow framework.

Link : https://www.udemy.com/course/deeplearning/

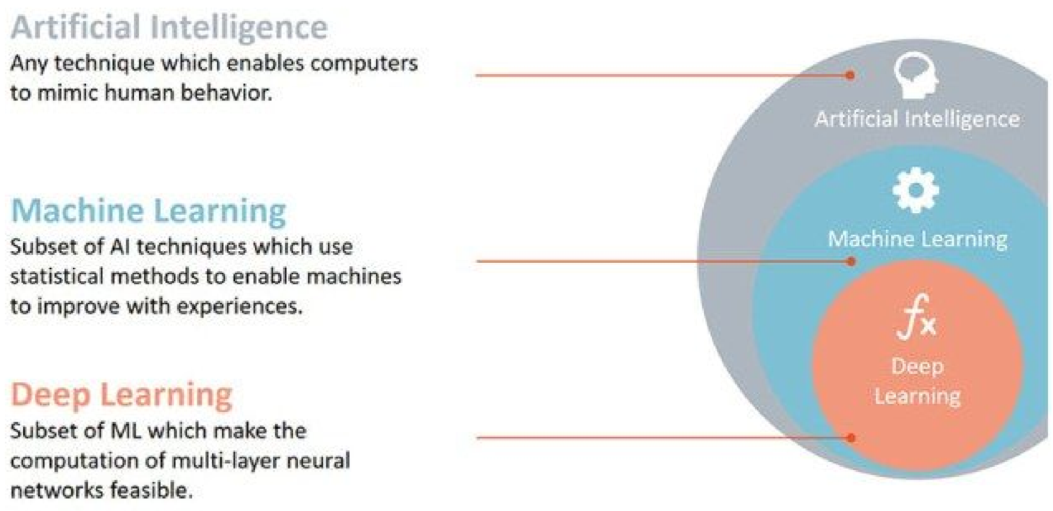
II. Fast.ai Practical Deep Learning for coders

Fast.ai is taught and made by Jeremy Howard (former President & Chief Scientist & top ranked performer on Kaggle, which is the top machine learning & data science competition platform). This course prioritizes practically first and concepts second, in contrast to the previous Deep Learning course I had done. It also utilizes its own Deep Learning library called fastai and runs on Pytorch instead of Tensor like what I was used to. This was written in Python as well.

Link : https://course.fast.ai/

What is Deep Learning ?

Deep Learning is a new tool within Machine Learning that's causing advancements in industries, and revolutionizing the capabilities of technology and what can be done with it. It’s ability to learn and teach itself from large data has proven it to be an impact on a multitude of industries. For example, self driving cars, virtual assistants, visual recognition, healthcare, fraud detection, and is even able to arguably create its own art.


Artificial Intelligence
Any technique which enables computers to mimic human behavior.
Machine Learning
Subset of AI techniques which use statistical methods to enable machines to improve with experiences.
Deep Learning
Subset of ML which make the computation of multi-layer neural networks feasible.
Artificial Intelligence
Machine Learning
Deep Learning

To understand Deep Learning one must know what a Neural Network is, because Deep Learning is at its bare bones - a Neural Network with many more hidden layers.

A neural network is a mathematical model used to discover patterns in data. This model was inspired by the human brain hence its name. A basic run-down of a neural network is this. The input goes into the Neural Network, is processed through the nodes (also referred to as neurons) in the hidden layers and the output comes out. An example could be for predicting sounds, a spoken word would be the input, and it would be processed through the nodes in the hidden layers. Then output its prediction.

Now, the difference between Deep Learning and Neural Networks is that Neural Networks only use a couple of hidden layers whereas Deep Learning uses many more layers. This enables Deep Learning to do both much more intensive processing on a large scale of data, allowing for more complex patterns to be predicted or formed.

//Deep Learning Run-Down

The real magic in Deep Learning happens in the approach it takes to process an output. It is still a neural network but more complex this time, even more like a human brain you could say. Every input has a node for it, and each one is given a 'weight' which is a value that will be altered appropriately after each epoch (round) of processing. The inputs then get sent to the first hidden layer nodes, the inputs to these hidden layer nodes are not necessarily a single input - it could be a combination of inputs (this is referred to as a feature). Now these hidden layer nodes analyze the received inputs and how they are put together through very complex mathematical formulas in order to reveal insights about the data, it then adjusts these inputs weights which may or may not be combined together in various ways and sends it to the next hidden layer. This process is repeated until it reaches the output. During training, this output would then be compared to the actual desired predictions to acquire an error rate. The model then uses this new information on what it got wrong to adjust all of its hidden layers simultaneously in an effort to get the desired output. This entire process is a simple explanation of one round of training, technically referred to as one epoch. In my opinion, this way of processing or 'learning' is very similar to a human brain. Because Deep Learning learns from experience and not hard-coded rules. This all results in Deep Learning being able to handle big data, and be very flexible in its applications.

Supervised Learning vs Unsupervised Learning

There are two main types of machine learning. Supervised and unsupervised learning.

Supervised :-

- Supervised learning is when you train a model using correctly 'labelled' data. Basically that within your data - the correct answer is known. This is so that at the end of each epoch it can compare the predicted result with the actual result. Acquire an error and then adjust itself in a way that would yield the desired result. Supervised learning entails basically either a classification (classify input into some class - e.g. scanning x ray for fracture) problem or a regression (to predict a continuous value - e.g. realestate price, or even a sports players makeshift rating).
- My 3 practical projects during this independent study were all supervised learning projects.

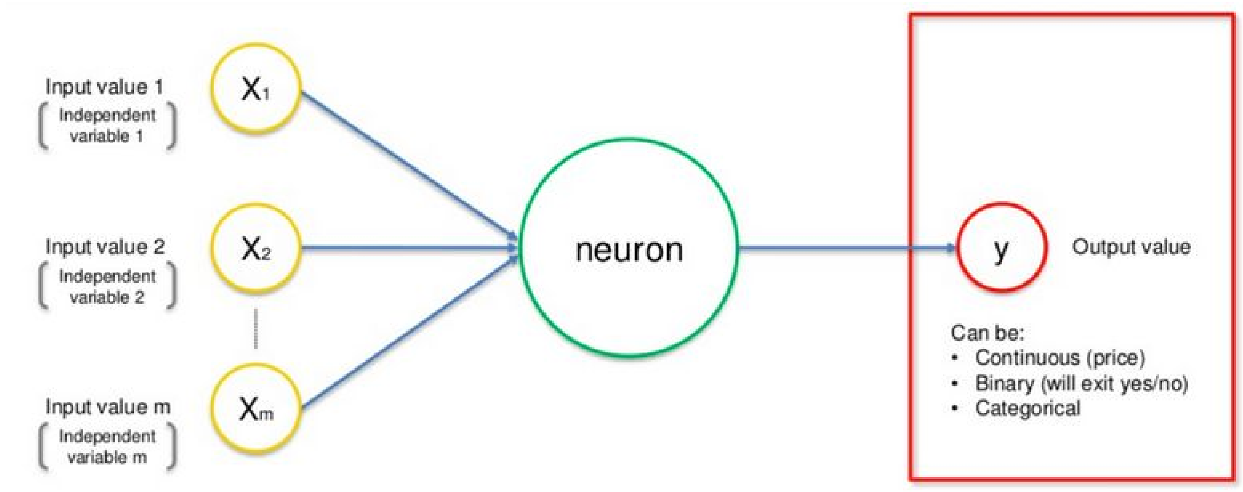
Unsupervised :-

- Unsupervised learning is basically when you do not 'supervise' the learning. I.e. You do not give it correctly 'labelled' data. Instead, the model receives unlabelled data that it needs to make sense of on its own. This type of learning might reveal insights in data that might otherwise have gone unnoticed. Unsupervised learning basically entails either a clustering (you want to discover groupings within the data - e.g. grouping customers by purchase history) problem or a association (you want to discover 'rules' that can organize large data - e.g. customers that bought Y, also tend to buy X).
- Note that you can create systems that utilize both types of learning. For example, a self driving car.

ANNS - Artificial Neural Networks :-

// Step 1 - Inputs & Neuron

Starting from the ground up - The neuron.

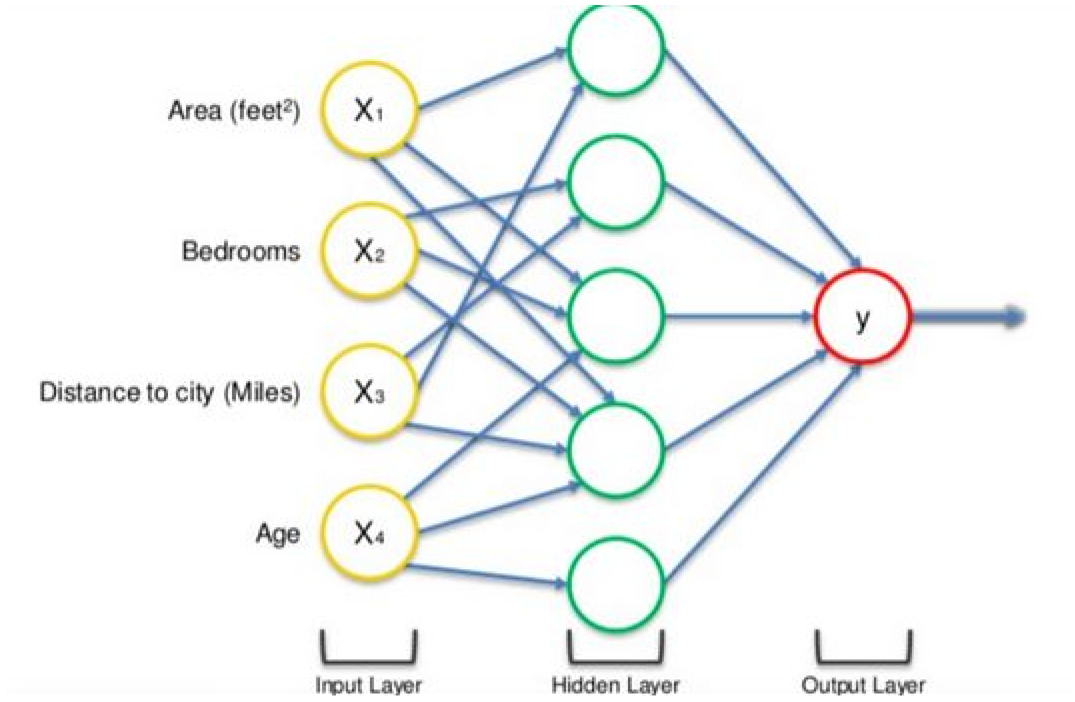


For every observation there are inputs that are independent variables which get processed through a neuron or neurons (depending on layers) and through those layers, the input is processed into an output or outputs. It is extremely important that in order to pre-process your data you have to either standardize or normalize them. Otherwise, your model will not be accurate and might train absurdly on some over-valued feature.

// Step 2 - Weights = Synapses

The weights can be seen as synapses between all the neurons. I.e. They are what bind the input values to the neuron and from one neuron to the next and so on. There is an individual weight assigned to each synapse, this weight determines how strong the signal that passes along that synapse is. I.e. It determines how much value that feature will have in the processing. This effectively gives the model flexibility in deciding what inputs go where, allowing the model to create unique combinations of features. Weights are instrumental in a Neural Networks learning, as it decides what is important and what isn't based on the weight. This is what is adjusted constantly through learning, and is the only thing that is changed in training.

An example of a 1 hidden layer neural network

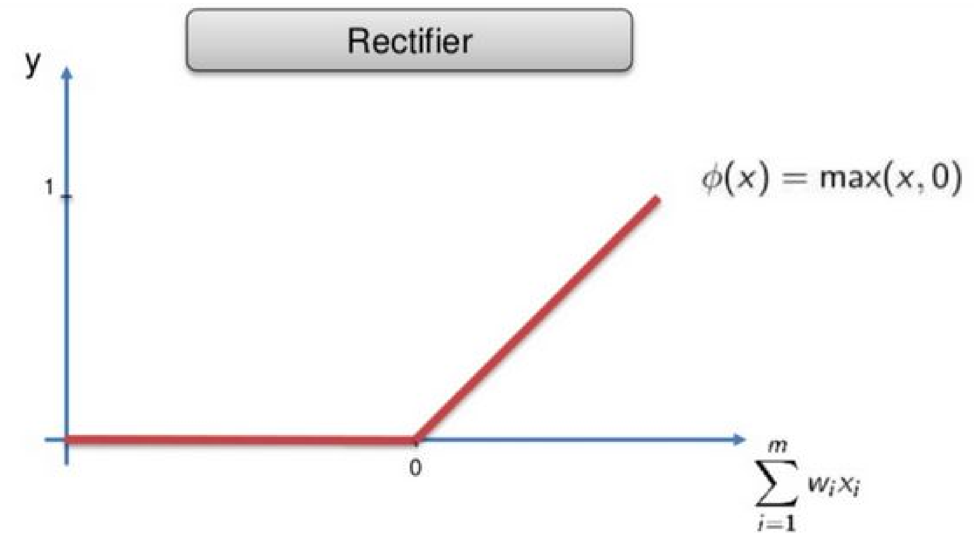


Note how each hidden neuron is receiving its unique set of inputs. E.g. The fourth hidden neuron is receiving all the inputs but the fifth hidden neuron is receiving only the age input.

// Step 3 - Activation Function in the Neurons

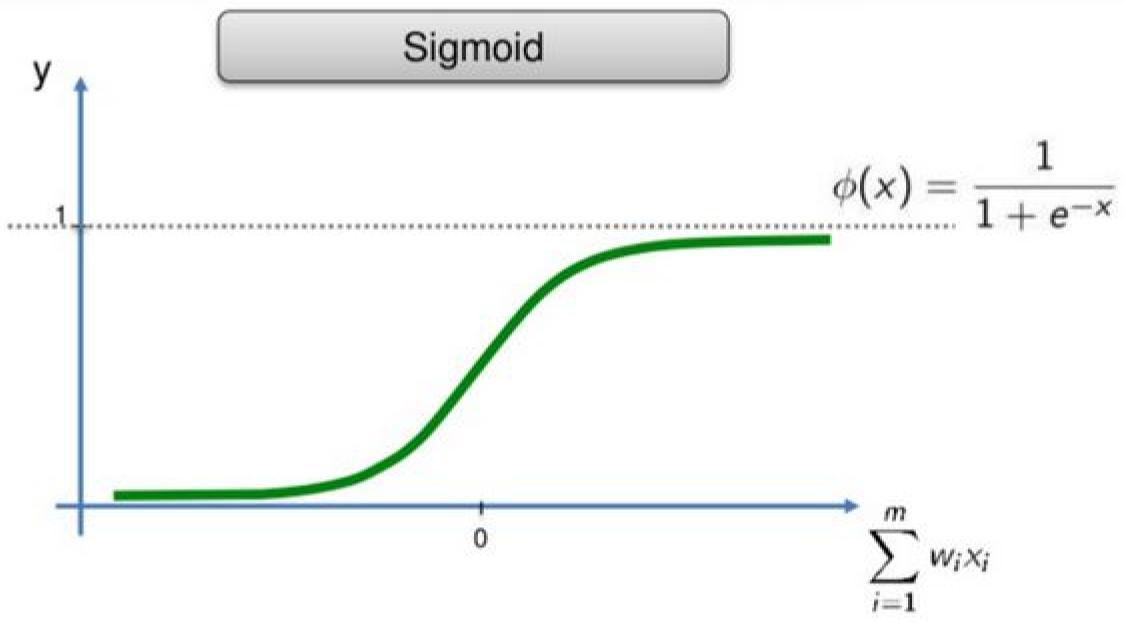
Now, the neuron adds up the sum of all the weights. It then takes this sum and applies the activation function that is corresponding to that neuron or that entire layer of neurons. These activation functions are diverse and are derived from heavy mathematical theory, but to be a good practitioner of Deep Learning doesn't require you to understand the complexities behind the curtains. To be a good practitioner of Deep Learnings requires you to familiarize yourself with the popular functions and know the differences of each one in order to choose the best one for your situation. This is why I will briefly go over some popular activation functions.

The Rectifier Function (ReLU)



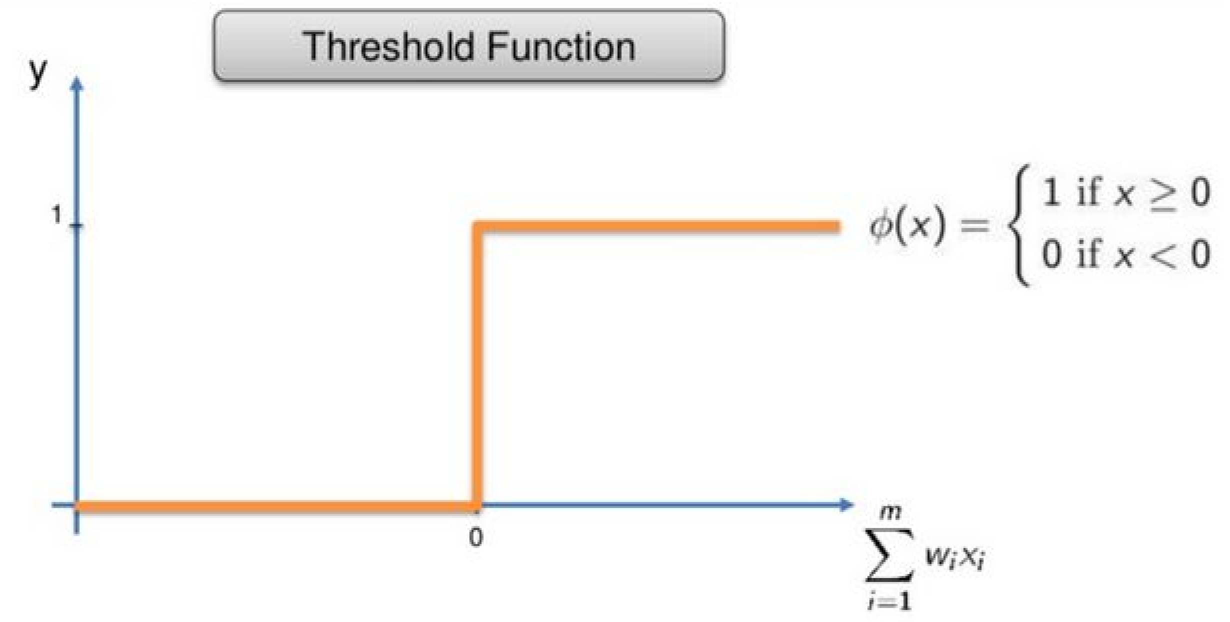
This function makes a network sparse, allowing it efficient and easy computations. It does so by eliminating any value less than or equal to zero. This means that not all the neurons get activated, thus making the network sparse.

The Sigmoid Function



This function is very popular and well suited for outputs that are probabilities. This is because the domain is literally limited to 0-1. If a value is above 1 it gets dropped to 1, if a value is below 0 it gets dropped off to 0. This function might be most commonly seen in the last layers or even output layer.

The Threshold Function



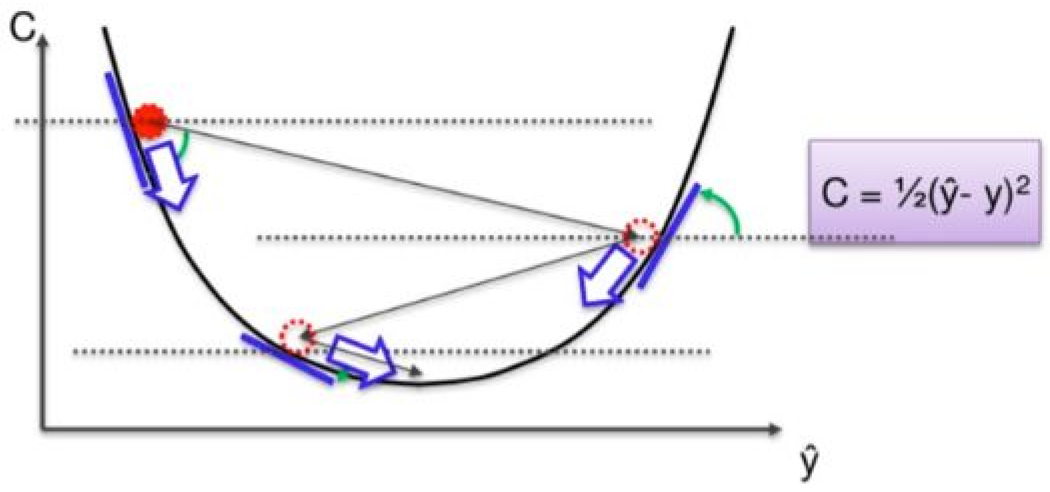
A very simple function, mainly used for binary outputs and found in the last layer or output layer. If 'x'/feature has any positive value then it is a 1, if not then it is a 0.

// Step 4 - Error/The Cost Function & Learning through Gradient Descent

The model, after running inputs through all the hidden layers and producing a value. It then compares this predicted value and the actual value to find the difference between the two. It then applies this value into a "Cost Function". A Cost Function basically tells us the error in our prediction. There are various types of Cost Functions suitable for different purposes.

The goal is to reduce this cost function as much as possible. This is done by utilizing a gradient descent algorithm, often referred to as 'optimizers'. The most popular one seems to be 'adam'. This process entails a backwards pass from the output layer to the input layer, this determines the gradient of the cost function with respect to each weight. Giving a grasp on which weight affects it, thus allowing impact-full weight adjustments. Using this slope it gained from the gradient, it adjusts its values accordingly so that it would be at the lowest point of error - i.e. the global minimum, where the slope is 0. For example if the slope is negative, then the algorithm would adjust it so that it kept going downward in an effort to find its lowest point, and if it over pushes and now the slope is positive - it would adjust the weights to bring back the slope the other way.

This is what happens at the end of every epoch (round of processing). The cycle will just continue for the desired number of epochs.



This is how a Gradient Descent algorithm might adjust its weights according to the gradient of the loss function (which was acquired through backpropagation) in order to end up in a minimum.

ANNS - PRACTICAL WORK :-

Below are my ANN practical coding notes that I made while creating an ANN that would predict whether a customer would leave the bank based on 10 features. This worked on top of tensorflow. Some methods I used during pre-processing were encoding categorical data into numerical value, creating dummy variables, fixing missing variables, scaling and splitting test/train set etc. To improve the model I used parameter tuning and the dropout method. I used a .csv file containing mock information on customers.

//ANN CODE WITH NOTES FOR COMPREHENSION

Step 1. Preprocess Data #always

Step 2. Creating our ANN i.e. classifier

Step 3. Compiling it via classifier.compile() and training it via classifier.fit(X_train, y_train, batch_size, epoch)

Step 4. Predicting using the IV_test and evaluating models performance.

PART 1 :- DATA PRE-PROCESSING

1. **Importing the dataset & exporting the IVs and DVs to our object arrays.**

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

dataset = pd.read_csv('Data.csv')
```

  - .read_csv(' mydatafile.csv ')
  - Imports our dataset.

```
X = dataset.iloc[:, :-1].values # X is always INDEPENDANT VARIABLES

y = dataset.iloc[:,-1].values # y is always DEPENDANT VARIABLES
```

  - Extracting our values. Designating which columns to extract from our data set according to our IVs and DV.

- **IF you need to encode categorical data, OR take care of missing data, YOU DO SO BEFORE SPLITTING!** *Look in preprocessing template for these shenanigans*

2. **NOW YOU CAN SPLIT IT INTO/Create OUR TRAINING & TEST SET.**

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state = 0) # Creates our X_train, y_train & X_test, y_test sets.
```

- train_test_split(<IV-object> , <DV-object> , test_size = ??)

3. **FEATURE SCALING!** ***# Always required! Scales all values to be within a suitable range. FASTER PROCESSING + NO OVERVALUED I.V***
    - Do not scale DV (y) if it is binary!

```
from sklearn.preprocessing import StandardScaler

sc_X = StandardScaler()
```

- Creating our StandardScaler object

```
X_train = sc_X.fit_transform(X_train)
```

- fitting our sc_X object to X_train and scaling X_train.
- We must ALWAYS fit_transform() our training sets!
- fit_transform(<array>) #transform = scale basically

```
X_test = sc_X.transform(X_test)
```

- Scaling X_test, sc_X has already been fitted to X_train and so is transforming on the same fit basis.

PART 2 :- MAKING THE CLASSIFIER / ANN and TRAINING IT!

1. **Creating the classifier - layers! # NOTE: Not all programs need deep networks i.e. layers.**

```
import keras # or whatever class you will use. e.g. LogisticRegression from
sklearn.linear_model <- No layers for this

from keras.models import Sequential

from keras.layers import Dense

classifier = Sequential( )  # Creating & Initializing
```

- Initializes / Creates our classifier - Defining it as a *SEQUENCE OF LAYERS.*

```
classifier.add(Dense(units = 6, activation = 'relu', kernel_initializer = 'uniform',
input_shape=(11,) )) # Input + 1st Hidden Layer
```

- *Units* = # of Nodes = 6 -> We have 6 nodes (neurons) in this layer.
- *Activation* = 'relu' -> using the Rectifier activation function
- *kernel_initializer* = 'uniform' -> randomly initializes weights to small numbers close to 0
- *input_shape* = (# of inputs,) or *input_dim* = 11 -> Declares inputs ***#COMPULSORY FOR THE FIRST HIDDEN LAYER - IT CREATES THE INPUT LAYER***

```
classifier.add(Dense(units = 6, activation = 'relu', kernel_initializer = 'uniform'))
# 2nd Hidden Layer
```

- The same as above, but does not need to know inputs! It has the hidden layer before it to know.

```
classifier.add(Dense(units = 1, activation = 'sigmoid', kernel_initializer =
'uniform')) # Output Layer
```

- Contains only 1 node because we have a categorical DV in binary. Just 2 categories, a 1 or 0.
  - What if I have multiple categories?
  - If we have 3 or more categories in our DV, that means we are using the **OneHotEncoded** method (meaning we transformed the 3 category DV column into 3 DummyVs -> 3 columns of binary output) and **thus our # of output nodes = # of categories (DV columns - DummyVs).**
    - We would also replace the 'Sigmoid' function with its multi-categorical version, the 'Softmax' function.
- We use the sigmoid function to achieve a probability output from our model!

2. **Compiling the ANN! - Basically applying Stochastic Gradient Descent on the whole network!**

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])
```

- *optimizer* -> The algorithm applied to find/optimize the weights based on the gradient of the loss function. The Stochastic Gradient Descent algorithm!
  - There are many optimizers, 'adam' is just a good efficient one.

        - *loss* -> The cost-error-loss function to be optimized to find the optimal weights.
            - Remember, applying the logistic regression loss function to a sigmoid function yields a logistic regression model from which we can see clear predictions.
                - This is normally applied to a binary DV, but can still be applied to a categorical DV with 2 or more classes. 'crossentropy'.
            - 'binary_crossentropy' is the logistic loss function applied to a binary output.
        - *metrics* -> The criterion used to evaluate the model. The optimizer also uses this criterion to improve the models performance after every observation/batch.
            - The 'accuracy' metric is most typically used. Note that you can use multiple metrics so that is why the input is in array form.

- **Fitting classifier to the Training Set i.e. Connecting the ANN to the set and training it, it learns the correlation between IVs and DV in training set! //Skip to this part if an ANN wasn't needed.**

```
classifier.fit(X_train, y_train, batch_size=10, epochs = 100) #for an ANN
```

        - *batch_size* -> # of observations before the ANN updates the weights. I.E 'trains' itself through gradient descent.
            - This is up to you, the A.I. artist.
        - *epochs* -> The # of epochs to be completed. An epoch is a whole round through a training set.
            - Also up to you.

PART 3: PREDICTIONS & EVALUATIONS

- **Predicting the Test Set results**

```
y_pred = classifier.predict(X_test)
```

        - Predicting the test set, using the predict() function. Remember Y is our DV (result), thus y_pred is our predicted DVs
        - Outputs a probability as our ANN is outputting the probability.

```
y_pred = (y_pred > 0.5)
```

  - Needed to transform y_pred into a boolean value of true or false. 1 > .5 && 0 <.5.
  - Needed for confusion matrix, since our true DV results are in boolean, the two arrays to compare must be of the same value type!
- **Evaluating performance w/ confusion matrix #NEW METHODS WILL BE LEARNED**

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)
```

  - Returns a 4x4 matrix. 1st Row = [Correct, Wrong] 2nd Row = [Wrong, Correct] -> (Correct + Correct)/ (Dataset count) = accuracy.

Testing ONE NEW observation on the model!

  - You have to :

1.Convert your observation into an array object w/ a row for each observation. i.e. A horizontal array.

2. Scale it!

3.Plug it in to predict()

  - Can be done in one line of code! #go python

```
New_ObservationPrediction = classifier.predict(sc.transform( np.array( [ [ 1st obs IVs
- Features here ] , [ 2nd row obs] ,  [etc] ] ) ) )
```

  - Again, if you want a boolean - binary prediction use:

```
New_ObservationPrediction = (New_ObservationPrediction>0.5)
```

PRACTICAL CODING NOTES ON PARAMETER TUNING:

Parameter Tuning :-

Is using a GridSearchCV object which uses a KerasClassifier (wrapped model - w/ same architecture as real model) that tries different combinations of hyperparamaters (parameters that are not optimized autonomously through training) that we had suggested through a dictionary, this find the best combination that yields the best results/accuracy:-

- We use the GridCVobject.best_params_ and the .best_score_ functions to return our best accuracy and its corresponding parameters.
- Uses the k-fold technique to test the accuracy.
    - K-fold is splitting the practice set in 10 and using a different one of those to test the model during the fitting, thus reducing overfitting, and creating a more robust model.

Step 1 - Libraries:-

```
from keras.wrappers.scikit_learn import KerasClassifier

from sklearn.model_selection import GridSearchCV

from keras.models import Sequential

from keras.layers import Dense
```

- Importing our libraries.

Step 2 - Defining our 'build function':-

```
def build_classifier(optimizer):

    classifier = Sequential()

    classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation =
'relu', input_dim = 11))

    classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation =
'relu'))

    classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation =
'sigmoid'))

    classifier.compile(optimizer = optimizer, loss = 'binary_crossentropy', metrics =
['accuracy'])

    return classifier
```

- Defining our 'build function' which builds the architecture of our ANN

Step 3 - wrapping' / assigning our 'build function' to a classifier object:-

```
classifier = KerasClassifier(build_fn = build_classifier)
```

- This creates an object of which we can enter into the arguments of our GridSearchCV() function.

Step 4 - Creating our parameter dictionary (Which will be the 'options'/ i.e. the shi it tries :-

```
parameters = {'batch_size': [25, 32],
              'nb_epoch': [100, 500],
              'optimizer': ['adam', 'rmsprop']}
```

- Specifies the DICTIONARY (keys and its corresponding values) of parameters to test out.
    - The keys are self explanatory.

Step 5 - Creating/Initializing our GridSearchCV object! :-

```
grid_search = GridSearchCV(estimator = classifier,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 10)
```

- Initializes the GScv object with the correct arguments
    - Estimator = Our NN model
    - param_grid = Our parameter dictionary
    - Scoring = Metric we are judging by
    - cv = # of k fold cuts

Step 6 - Fitting the training set to our GScv object! :-

```
grid_search = grid_search.fit(X_train, y_train)
```

- Just like how we fit & train every other model.
- This will take a fat minute.

Step 7 - FEEDBACK! (Best accuracy and itst corresponing parameters :-

```
best_parameters = grid_search.best_params_
best_accuracy = grid_search.best_score_
```

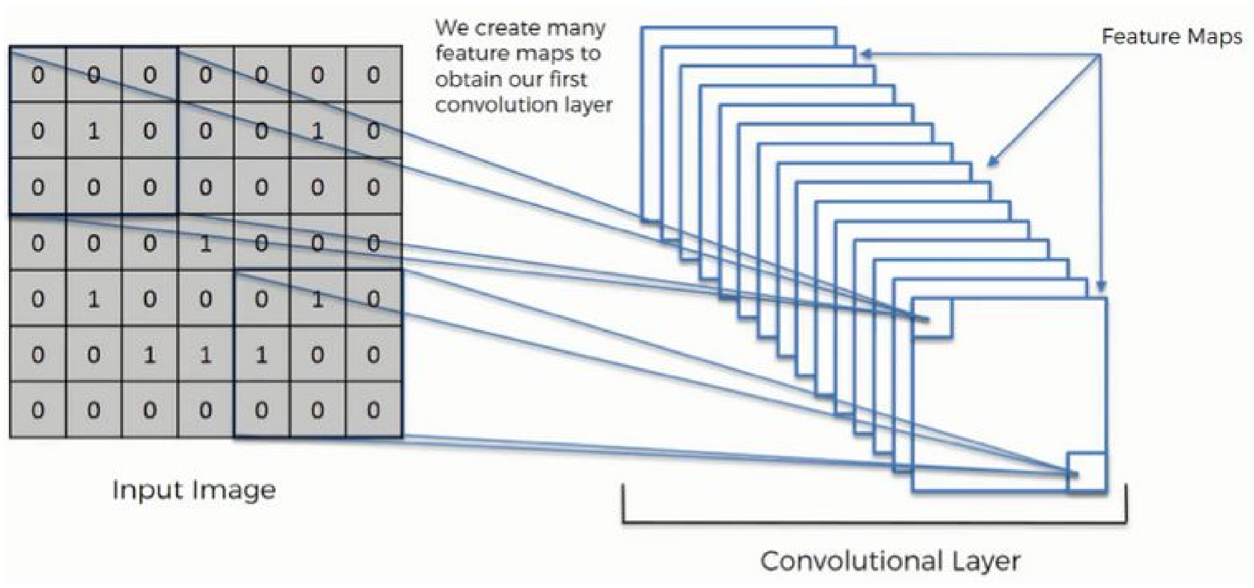
CNNs - Convolutional Neural Networks :-

A Convolutional Neural Network is similar to an ANN but each observation is an image, and each input value is a pixel value. It uses these pixel values to determine what's in the picture. These models are used a lot in the world, they allow self driving cars to identify objects and also how facebook can recognize faces.

The process can be explained in 4 stages

// Step 1 - Convolution : Filtering through a feature detector into a feature map & ReLU activation function

This stage involves modifying the original input image into a smaller image. It is important to note that these images are actually matrices of data and so I will refer to them as such. Convolution stage necessitates the use of a feature detector, also known as ‘kernel’ or ‘filter’. This feature detector is a much smaller matrice (usually 3x3, 5x5, 7x7) that is used to filter through the input image to acquire a smaller image called the feature map. This feature map is a smaller condensed image that contains only the most important parts/features of the original input. This smaller matrix that contains valued data now allows faster processing, more accuracy, and less overfitting. A convolutional layer comprises of multiple feature detectors, which results in multiple feature maps. The feature detectors are equivalent to the weights here, the values within the detectors are optimized by the gradient descent algorithm at the end of each cycle.


We create many feature maps to obtain our first convolution layer
Feature Maps
Input Image
Convolutional Layer

The process of filtering an input image through a feature detector to get a feature map:

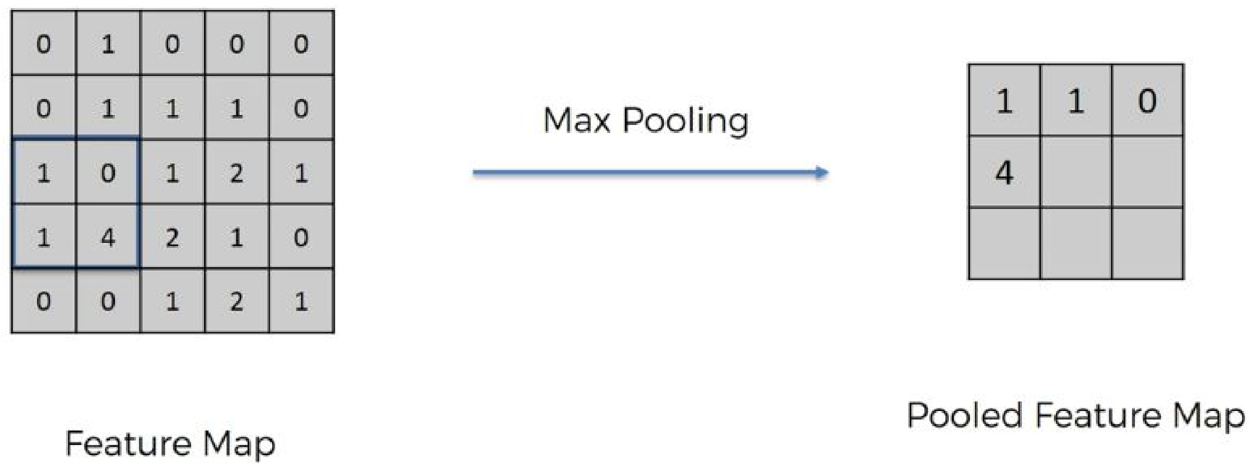
1. This 'features detector' i.e. digital matrice is used to filter through an input image. Counting the # of cells in which the feature detector matches the input image.
2. The # of matching cells is then inserted in the the top-left cell of the feature map.
3. The feature map then moves a # of cells to the right (the stride) and repeats the same process.
4. This is then repeated for every row, thus creating a 'Feature Map'.

In common practice, the rectifier activation function is then added on top of the feature map layer. This is to increase non-linearity in images. Linearity in images is the transition between pixel values, so to increase non-linearity means to differentiate the values moreso. This further highlights the important values in an image/matrix.

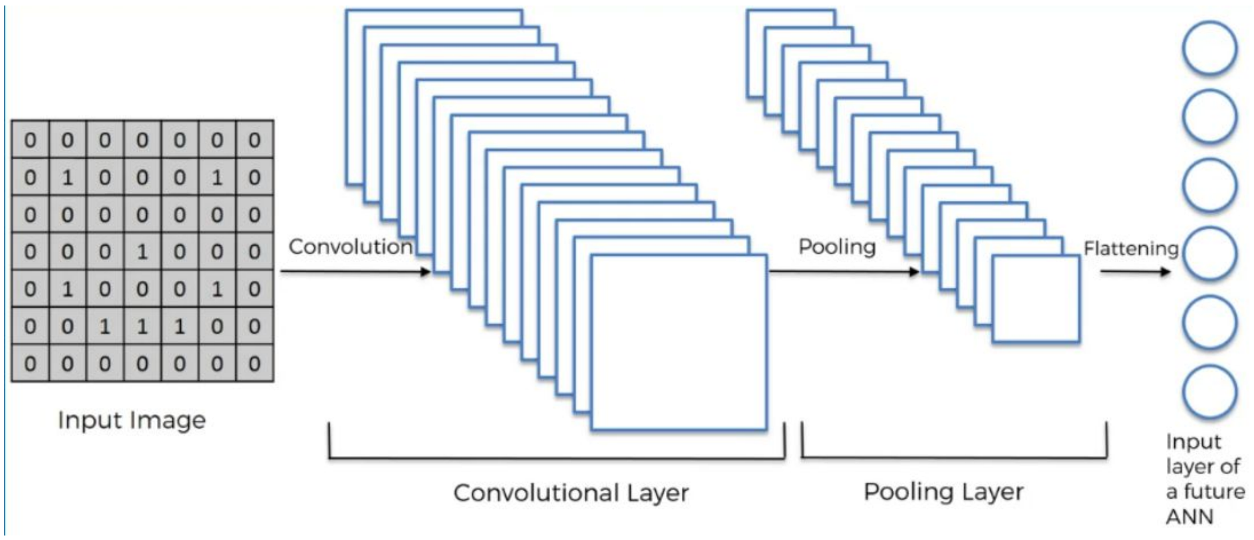
//Step 2 - Max Pooling to develop spatial variance

The purpose of max pooling is to enable the CNN to detect an object when it is presented that said object in any type of manner. This is called “spatial variance”. Spatial variance is the property that makes the network capable of detecting the object in the image without being confused by differences in texture, distance, angles, etc.

Max pooling is the process of filtering through an even smaller matrix (usually a 2x2 matrix) than the feature detector. This max pooling filter does not have any values to match like a feature detector does. Instead, max pooling scans on top of a section of the feature map and only takes the highest value This filtering also moves in designated 'strides' like a feature detector. This process results in a "Pooled Feature Map" which is now an even smaller and more valuable piece of data than the feature map. The diagram below illustrates how max pooling behaves.

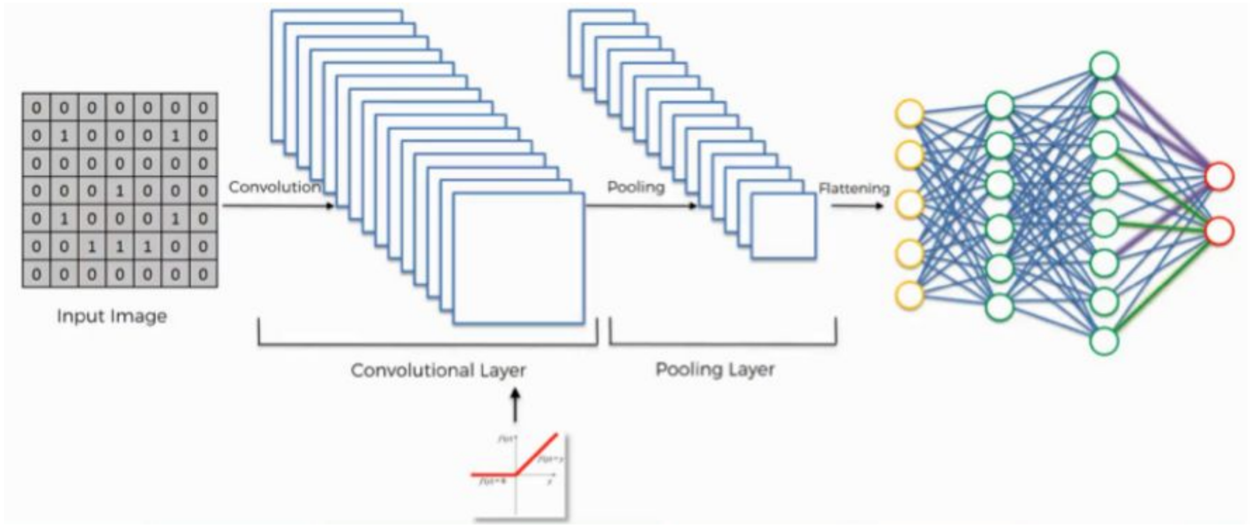


// Step 3 - Flattening the matrices into an input vector/layer & fully connected ANN



You now have multiple pooled feature map layers containing important pixel values, so you flatten this matrix into a vector (i.e. a column) so that this will be the input layer.

Now we add a full ANN structure as the next layer. But the key note is to create an ANN model that has all the neurons connected fully in the hidden layers. This means that no feature will get dropped, and that is important because our input is already of high value. This ANN layer is what will perform gradient descent and optimize the feature detectors at each epoch to train itself for a better accuracy.



An example CNN architecture.

CNN - PRACTICAL WORK:

Below is the practical CNN coding notes I made while creating the CNN. It's a CNN that runs on Keras on top of Tensorflow, it was built to classify cats and dogs. I used a dataset of 5000 images each for cat and dog. Some methods other than the usual pre-processing methods I used were - data augmentation as I only had 5,000 images each of cats and dogs. Multiple layers of Convolution + Maxpooling to reduce size while retaining valuable data (faster processing and better spatial variance), doubling # of feature detectors in consecutive conv layers. Building a fully connected ANN on top of CNN for better training.

I also created another CNN on Kaggle. Kaggle is a Machine Learning / Data Science website that hosts data science competitions. The competition was to build an accurate digit recognizer for the MNIST dataset. I built a CNN utilizing fastai instead of Keras this time and it yielded great results. I reached a high accuracy of 99.7%. The main reason for the high accuracy I believe is the transfer learning technique using fastAI utilizing resnet34 CNN model architecture and the learning rate finder method I learned from fastai. I also had to pre-processed by data wrangling (.csv files had to be .jpg) and restructuring arrays. Made an effort to incorporate more data visualizations, and be more neat in my notes. This was done in an online notebook environment similar to a Jupyter notebook.

This entire project is on kaggle through this link:
https://www.kaggle.com/yousefalkafif/digit-recognizer-mnist-using-fastai

```
# Part 1 - Building the CNN


# Importing the Keras libraries and packages
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense


# Initialising the CNN
classifier = Sequential()
```

```
# Step 1 - Convolution
classifier.add(Conv2D(32, (3, 3), input_shape = (32, 64, 64, 3), activation = 'relu'))

# Step 2 - Pooling
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Adding a second convolutional layer
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Step 3 - Flattening
classifier.add(Flatten())

# Step 4 - Full connection
classifier.add(Dense(units = 128, activation = 'relu'))
classifier.add(Dense(units = 1, activation = 'sigmoid'))

# Compiling the CNN
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])
```

# Part 2 - Fitting the CNN to the images

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)

test_datagen = ImageDataGenerator(rescale = 1./255)

training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                 target_size = (64, 64),
                                                 batch_size = 32,
                                                 class_mode = 'binary')

test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size = (64, 64),
                                            batch_size = 32,
                                            class_mode = 'binary')

classifier.fit_generator(training_set,
                         steps_per_epoch = 8000,
                         epochs = 25,
                         validation_data = test_set,
                         validation_steps = 2000)

# Part 3 - Making new predictions

import numpy as np
from keras.preprocessing import image
test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg', target_size
= (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_set.class_indices
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'
```

DATASET STRUCTURE & Pre-Processing:-

- Not a .csv data file.
- Use keras.

- We split the data ourselves by using folders.
  - Training:-
    - dataset\training_set\cats\cats1.jpg #up to 4000 so 4000 images
    - dataset\training_set\dogs\dogs1.jpg #up to 4000 so 4000 images
  - Test:-
    - dataset\test_set\cats\cats4000.jpg #up to 5000 so 1000 images
    - dataset\test_set\dogs\dogs4000.jpg #up to 5000 so 1000 images
- We scale it using:-

```
from keras.preprocessing.image import ImageDataGenerator


train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)


test_datagen = ImageDataGenerator(rescale = 1./255)
```

PART 1 -> BUILDING THE CNN

Importing the Keras libraries and packages & Initialization:-

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
```

# Initialising the CNN

```
classifier = Sequential()
```

- Declares 'classifier' as a sequence of layers.

Step 1 - CONVOLUTION LAYER :-

```
classifier.add(Conv2D(32, (3, 3), input_shape = (32, 64, 64, 3), activation = 'relu'))
```

- First argument is filters -> 32 ***You double the filters in every consecutive layer**
    - The # of filters. -> Also our # of feature maps obviously.
- Second argument is the (Height, Width) -> (3, 3)
    - Our (height, width) of our matrix. Here it is a 3x3 matrix.
- input_shape argument is the (Batch Size, Rows/Height, Columns/Width, Channels) -> (32, 64, 64, 3)
    - *Our input specifications of our images*. i.e. what our conv layer should expect to receive.
        - NOTE: We have to resize the images during our importation of the image folders in the fitting stage.
    - (batch size, height, width, channels)
- activation argument -> 'relu'
    - Our activation function.
    - Using the rectifier to eliminate negative value to reduce linearity
    - This is done to make up for the linearity we might have imposed through the process of creating a FM.
- Padding argument. #to be researched
    - Padding is when you add '0's surrounding the image so that the matrix can fully scan the image.

- $$\begin{bmatrix} x & x & x & o & o & ? \\ x & x & x & o & o & ? \\ x & x & x & o & o & ? \\ x & x & x & x & x & \\ x & x & x & x & x & \end{bmatrix}$$ -------> W/padding

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & x & x & x & o & o & o & 0 \\ 0 & 0 & x & x & x & o & o & o & 0 \\ 0 & 0 & x & x & x & o & o & o & 0 \\ 0 & 0 & x & x & x & x & x & 0 & 0 \\ 0 & 0 & x & x & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

NOTE: It is better to have multiple (Conv + Pooling) layers stacked on top of each other. DEEP learning broski.

- ***You typically double the amount of filter detectors in every consecutive layer.***

NOTE: The bigger the image the larger the stride, otherwise it would take too long.

Step 2 - POOLING LAYER:-

```
classifier.add(MaxPooling2D(pool_size = (2, 2)))
```

- pool_size -> Our (Width, Height) of pooling matrix

Step 3 - FLATTENING:-

```
classifier.add(Flatten())
```

- Self explanatory

Step 4 - ADDING A FULLY CONNECTED ANN on top:-

```
classifier.add(Dense(units = 128, activation = 'relu'))

classifier.add(Dense(units = 1, activation = 'sigmoid'))
```

- There is no set method of finding the right amount of nodes. Just remember that we are having a ton of inputs (the maps pixels/values) flattened.

Step 5 - COMPILING THE CNN:-

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = 
['accuracy'])
```

- We normally use crossentropy for CNNs classification problems.
  - binary_crossentropy for a two class problem. e.g. cat or dog.

PART 2 -> FITTING THE CNN TO THE IMAGES

# All taken from keras.io\preprocessing

Import:-

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

Step 1 - Setting up ImageDataGenerator object (for Feature Scaling + Data Augmentation)

```
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2, # to manipulate images in batches 
- prevents over fitting
                                   horizontal_flip = True)


test_datagen = ImageDataGenerator(rescale = 1./255)
```

- Rescale argument

    - the feature scaling - scaling all our pixel values between 0-1. e.g. 255* 1/255 would be a 1 <- thats the max
- Shear_range + zoom_range + horizontal_flip and optional others:-
    - # data augmentation settings to manipulate images in batches.
    - Prevents over-fitting
    - Allows us to re-use the same images but in different variations. So we can use 'small' data sets.
- NOTE:
    - Test dataset does not get augmented. We are just validating the model after all.

Step 2 - Importing dataset using ImageDataGenerator object:-

```
training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                 target_size = (64, 64),
                                                 batch_size = 32,
                                                 class_mode = 'binary')
test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size = (64, 64),
                                            batch_size = 32,
                                            class_mode = 'binary')
```

- First argument is our directory.
    - our directory, make sure it is in your working directory.
- target_size argument:-
    - converting our images to the same size that is EXPECTED IN our CNN model.
- batch_size argument:-
    - size of batch in which the random samples of our images will be included.
- Class_mode argument:-
    - One of "categorical", "binary", "sparse".

Step 3 - Fitting & Training our model! :-

```
classifier.fit_generator(training_set, # training set argument
                         steps_per_epoch = 250, # total number of batches to be done before declaring an epoch to finish, i.e. batches per epoch - should be typically -> samples in dataset/batch size
                         epochs = 25,
                         validation_data = test_set,
                         validation_steps = 63) # batches per epoch
```

- First argument
  - Is our training set object.
- steps_per_epoch argument :-
  - The amount of batches per epoch.
  - Should typically be (samples in our set)/(batch size)
- validation_data argument :-
  - Our test set object
- validation_steps argument :-
  - Just batches per epoch.

HOW TO TEST MODEL ON ONE NEW SINGLE IMAGE :

```
import numpy as np
from keras.preprocessing import image
test_image = image.load_img('dataset/single_prediction/cat_or_dog_1.jpg', target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_set.class_indices
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'
```

Importing Numpy:-

- To use a function that will preprocess the image, so that it will be accepted by our predict() function

- Neural Networks .predict() always expect a 4D array object with one dimension corresponding to the batch so we have to add an extra dimension to our 3D image.
    - We can have multiple batches with different amounts of data in them. 1 batch might contain 2043 images whilst another contains 1 etc.
    - Using .expand_dims(test_image, axis = 0)
    - Our added dimension is the batch.
    - Our object must be an array! # use .img_to_array() beforehand.
    - axis -> Specifies the index position of the new dimension.
        - NOTE: I think using ImageArrayObject[np.newaxis] will create the 4th dimension for the whole array object.

Importing image module from keras.preprocessing:-

- To import/load/assign image to object:-

```
image.load_img()
```

    - Should use the same dimensions.
- To convert image to array with:-

```
image.img_to_array()
```

    - Just input our image which is a variable now from the .load_img()
    - This would convert it into the correct float32 type 3D array for colored images. The size dimensions are kept the same but it adds the RGB dimension e.g. 64x64 -> 64x64x3

Predicting our result with .predict():-

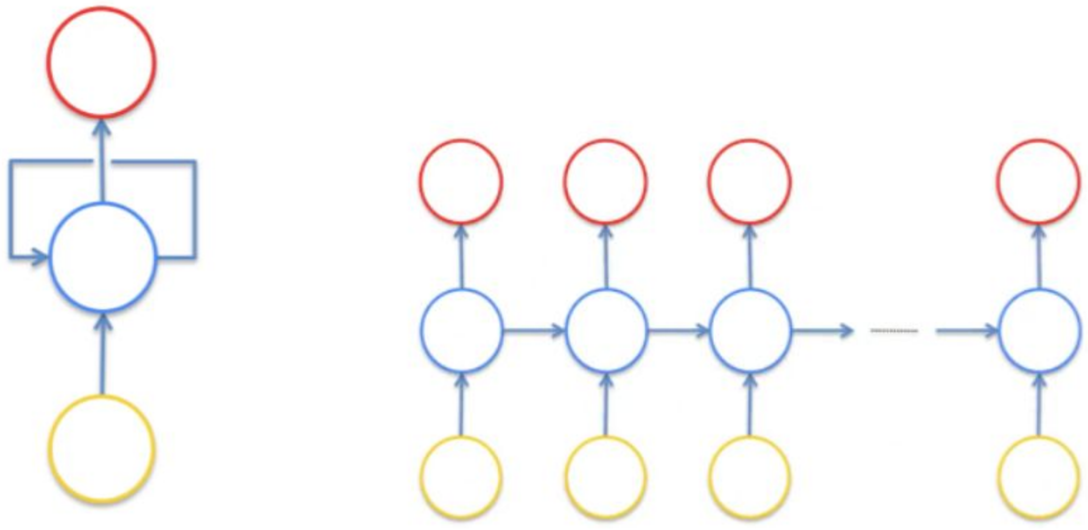
- result = classifier.predict(image4Darray)
    - Will return a number.
- To decipher what this number means we use .class_indices on our training set
    - training_set.class_indices -> returns on console a dictionary specifying what the number correlates too. # { 'keyIDentifier' : 0, 'Id2' : 1, etc }

- We can then use this to create a logic code segment that will print our answers.
    - if result [0][0] == 1:
        - Needs no explanation
        - We use [0][0] because the .predict() returns a 2D array.

RNN - Recurrent Neural Networks

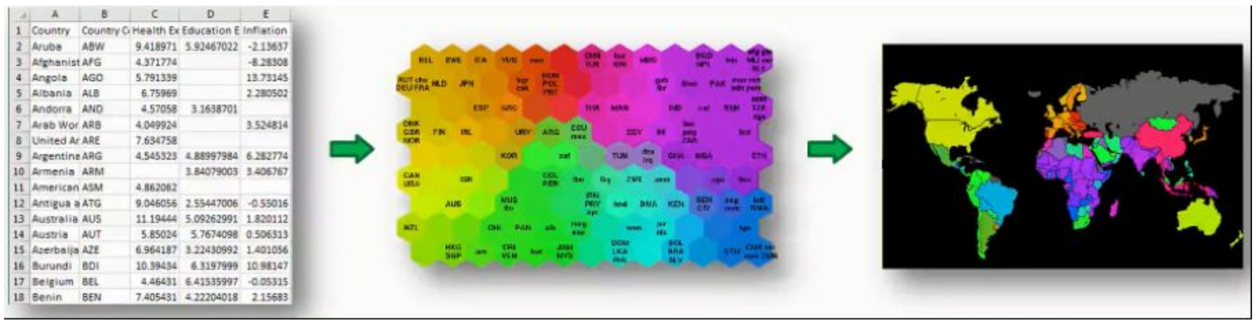
The main idea behind Recurrent Neural Networks is that the neurons can pass information back onto themselves in the future, as a form of short term memory so that the model could make a more appropriate prediction. This model is very useful for captioning movies or translating languages. As these actions require contextual information to be performed appropriately. The RNN accomplishes this by having its hidden layers loop back onto themselves and also produce an output at the same time, like so.

The image on the left and on the right are the same diagram, just two different views.



SOMs - Self Organizing Maps

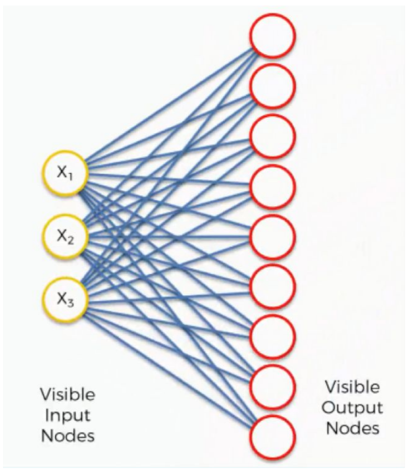
Self Organizing Maps (SOMs) is a form of unsupervised learning. This mean that the input data is unlabelled and the model has to make sense of the data and organize it by itself. It has to learn and classify the datas components on its own. The main feature that SOMs accomplish is to reduce dimensionality. Reducing dimensionality means that it takes data that has hundred of columns and rows, and processes it into a simplified map.

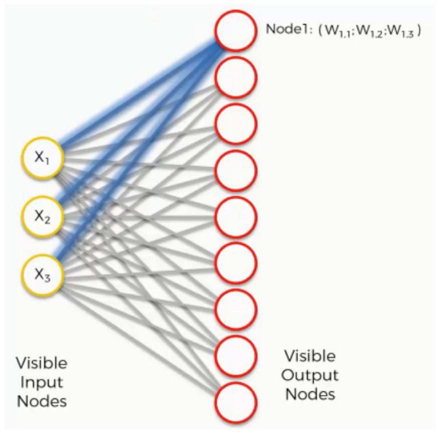


	A	B	C	D	E
1	Country	Country C	Health Ex	Education E	Inflation
2	Aruba	ABW	9.418971	5.92467022	-2.13637
3	Afghanist	AFG	4.371774		-8.28308
4	Angola	AGO	5.791339		13.73145
5	Albania	ALB	6.75969		2.280502
6	Andorra	AND	4.57058	3.1638701	
7	Arab Wor	ARB	4.049924		3.524814
8	United Ar	ARE	7.634758		
9	Argentine	ARG	4.545323	4.88997984	6.282774
10	Armenia	ARM		3.84079003	3.406767
11	American	ASM	4.862062		
12	Antigua a	ATG	9.046056	2.55447006	-0.55016
13	Australia	AUS	11.19444	5.09262991	1.820112
14	Austria	AUT	5.85024	5.7674098	0.506313
15	Azerbaija	AZE	6.964187	3.22430992	1.401056
16	Burundi	BDI	10.39434	6.3197999	10.98147
17	Belgium	BEL	4.46431	6.41535997	-0.05315
18	Benin	BEN	7.405431	4.22204018	2.15683

This is an example of a dataset that only contained countries and 39 other features of the country, it took the dataset and organized it into clusters that represented a spectrum of well-being. This SOMs insight could then be translated into a better form of data visualizing.

This is a basic example of an SOMs architecture.





Notable differences between an SOM and a ANN:

- Weights are not part of a computation in the nodes, but rather act as the output nodes coordinates (shown on the right diagram)
- Input nodes are not all features of one observation, but rather each input node is a dimension/column of the input, and this dimension may contain any number of rows.

Technical Tools / Comments :-

Tensorflow and Pytorch are the two most popular open-source frameworks for deep learning. They are both built on Python. Tensorflow is arguably the more industrial dominant one, it was developed by Google and is used in google search, their speech recognition system, and much more of their services. Other companies using Tensorflow include Uber, Intel, Ebay, AirBnB and more.

You can use tensorflow on Google Collaboratory or install it on your system. For my practical purposes I chose to install it using Anaconda which is a data science platform environment manager that was highly recommended.

I used mainly the Keras API, which is written in Python and runs on top of TensorFlow. Keras was developed with a focus on fast experimentation and I found it to be very user friendly once I was familiar with the terms. I utilized this for both my ANN customer churning project and my CNN dog/cat classifier project.

I also dabbled in fast.ai which is a very new Deep Learning library that runs on Pytorch. This library and course was very much results oriented and I preferred fast.ais library moreso than Keras, but they both contain the same language so there is little difference between them. I utilized fast.ai on a Kaggle competition for recognizing hand-written digits from the MNIST dataset. I managed to get an accuracy of 99.7%.

Conclusion :-

Overall, I feel like my independent study on deep learning has given me insights and a certain level of technical familiarity in a field that is only going to grow. The concepts behind it all are very intuitive and interesting. The practical work of coding the models was not as daunting as it had seemed at first, it took a while to fully quite understand and I had to write out notes as I went along for comprehension purposes. I am glad I wrote all those notes because they've enhanced my comfortability with the technical jargon involved in deep learning. This independent study hopefully is just the beginning of my Deep Learning career as I plan to continue learning about it.

Practical Work Links (notes are included in the links) :

Kaggle - CNN Digit Recognizer (MNIST Dataset) Competition utilizing Fastai ontop of Pytorch

Link: https://www.kaggle.com/yousefalkafif/digit-recognizer-mnist-using-fastai

Most notable remark: The main reason for the high accuracy I believe is the transfer learning technique using fastAI utilizing resnet34 CNN model architecture and the learning rate finder method I learned from fastai. Pre-processed by data wrangling (.csv files had to be .jpg) and restructuring arrays. Made an effort to incorporate more data visualizations.

GitHub - CNN project using Keras ontop of TensorFlow

Link : https://github.com/usefkafif/DogCatCNN

Most notable remark: A CNN I built to classify cats and dogs. Some methods other than the usual pre-processing were - data augmentation as I only had 5,000 images each of cats and dogs. Multiple layers of Convolution + Maxpooling to reduce size while retaining valuable data (faster processing and better spatial variance), doubling # of feature detectors in consecutive conv layers. Building a fully connected ANN ontop of CNN for better training.

GitHub - ANN project using Keras ontop of TensorFlow

Link : https://github.com/usefkafif/CustomerChurnANN

Most notable remark: An ANN I created to predict whether a customer will leave bank based on 10 features. Some methods I used during pre-processing were encoding categorical data into numerical value, creating dummy variables, fixing missing variables, scaling and splitting test/train set etc. To improve the model I used parameter tuning and the dropout method.

Please note that for the github projects - there are more detailed practical notes within this document and not on the github link.