SAHAAY
Social Innovation Club

PRABHAAV

DESIGN TO GENERATE IMPACT

CFI
SUMMER
SCHOOL
2022

Programming
Module

Numpy

```
import numpy as np
```

Why Numpy?

Because Numpy `ndarrays` work much much faster than Python `lists` both in storage and operations over vectors. Also an added benefit - if this is Vanilla Python:

```
a = range(0,10,1)
b = range(1,11,1)
c = [(a[i] + b[i]+5)*2 for i in range(10)]
a[1:3]=[-1,-1]
```

This is the much more convenient Numpy equivalent:

```
a = np.arange(0,10,1)
b = np.arange(1,11,1)
c = (a+b+5)*2
a[1:3] = -1
```

Array Declaration & Attributes

Shape of an array is represented as a tuple. The left most entry represents the highest nesting. For example, `np.array([[1,2,5],[3,4,6]])` has the dimensions (2,3). Consider the list `a=[1,2,5,3,4,6]` and `shape = (2,3)`, Below are a few ways to declare arrays in Numpy and the corresponding arrays declared:

- `np.array(a)` » array([1,2,5,3,4,6])
- `np.arange(0,3,0.5)` » array that starts a 0, steps up each element by 0.5 until it's lesser than 3 (does not include 3).
- `np.full(shape,2)` » array([[2,2,2],[2,2,2]])
- `np.linspace(0,3,7)` » array with 7 equally spaced elements starting from 0 till 3 (both included)

Below are few attributes we may use to retrieve info from an array. Let the array be `b=np.array(a)` here:

- `b.ndim` » Dimensions of `b` ie 2. It's the number of nestings.
- `b.reshape(shape)` » array([[1,2,5],[3,4,6]]), b remains unchanged.
- `b.shape` » (2,3)
- `b.size` » 6, total number of elements
- `b[2:4]` » array([2,5]), note that slice is a shallow copy in Numpy

Broadcasting

Consider the example below that is very similar to the one in the first slide. Let us visualize 2d-arrays as matrices for now.

$$\begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} + 5 = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 5 & 5 \\ 5 & 5 \\ 5 & 5 \\ 5 & 5 \\ 5 & 5 \end{bmatrix} = \begin{bmatrix} 5 & 6 \\ 6 & 7 \\ 7 & 8 \\ 8 & 9 \\ 9 & 10 \end{bmatrix}$$

Here, the way the 5 which is has an `ndim` 0 gets expanded to the shape $(5,2)$ is an example of Broadcasting.

Steps for Broadcasting: Let us say we need to broadcast `np.arange(3)` and `np.arange(3).reshape((3,1))` for addition.

1. Keep putting `1` to the left side of the shape-tuple until all the addends reach the `ndim` of the one with the highest. In our arrays, the shape of the first array $\begin{bmatrix} 0 & 1 & 2 \end{bmatrix}$ goes from $(3,) \rightarrow (1,3)$ to match the highest `ndim` which is of the second array. Visually, there's clearly no change.

2. Expand all 1s in shape-tuples by duplicating the elements along that axis to match the highest size in that dimension.

$$\begin{bmatrix}0 & 1 & 2\end{bmatrix} + \begin{bmatrix}0\\1\\2\end{bmatrix}$$

$$\implies \begin{bmatrix}0 & 1 & 2\\0 & 1 & 2\\0 & 1 & 2\end{bmatrix} + \begin{bmatrix}0 & 0 & 0\\1 & 1 & 1\\2 & 2 & 2\end{bmatrix}$$

$$\implies \begin{bmatrix}0 & 1 & 2\\1 & 2 & 3\\2 & 3 & 4\end{bmatrix}$$

Thus our first array goes from $(1,3) \to (3,3)$ since the highest in the first axis is 3. The second array goes from $(3,1) \to (3,3)$ since the highest in the second axis is 3. Now addition compatible.

Note that if two arrays can't be made to the same shape following the above procedure, then the two can't be added.

Pandas

```
import pandas as pd
```

Intro to Pandas

It's a spreadsheet module. The commonly used structures in Pandas are:

- `pd.Series([64,2,8],index=["A","B","C"])` » A column of entries
- `pd.DataFrame(data_dict)` where
  data_dict is a Dictionary of Column Headers as Keys and `pd.Series` as values
  or data_dict is a dictionary with Column Headers as keys and lists as values (ordered accordingly). The row index will become 0,1,...
  or data_dict is a nested list. The Column Headers will become 0,1,... too.

Pandas let's us import and export onto CSV files

- Importing: `input = pd.read_csv(r"/folder/folder 2/input_data.csv")`
- Exporting: `output.to_csv(r"/folder/folder 2/output_data.csv")`

Pandas also let's us retrieve data using row index and column header.

- Using Column Header:
  `dataframe_name["column name"]` » `pd.Series` of index and the intended column.
  `dataframe_name["column name"].values` » A Numpy array of all values in the `pd.Series` that was returned above.
- Using Integer Row Index:
  `dataframe_name.iloc[a]` where
  - a is a single integer like `1`: returns a `pd.Series` of that row with column headers as index. `.values` attribute of this gives a Numpy array as in the Column Header case.
  - a is a slice object like `1:2`: returns a `pd.DataFrame` with the specified rows alone. `.values` attribute of this gives a 2D Numpy array with the each row as one sub-array.
- Using Both Integer Rows Index & Integer Column Headers:
  `dataframe_name.iloc[0:1,2:4]` » returns a `pd.DataFrame` with rows corresponding to the first slice object and columns to the second.
- Using a Comparison Constraint:
  `dataframe_name[dataframe_name["column_name"]>1]` » Dataframe of all rows with value in "column_name" greater than 1.

Visualizing Data

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

Knowing to visualize data in graphs will aid us compare our results with what is roughly expected. Here is a code to plot a line graph and a scatter plot with legends in the same axes and then display a fresh new line graph without the previous plot. but only the new one.

```
x=np.arange(6)
y1=np.linspace(0,10,6)
y2=[0.1, 2.1, 3.8, 6, 9, 9]

plt.plot(x,y1, label="Line Plot")
plt.scatter(x,y2, label="Scatter Plot")
plt.legend()
plt.show()

plt.plot(y1,y2)
plt.show()
```

Note that both `ndarrays` and `lists` are okay here. Further, the Graph and Axes may be labelled using `plt.xlabel("label")`,`plt.ylabel("label")` and `plt.title("title")`

Some more Numpy and Pandas stuff to keep in mind:

- Sometimes the DataFrame (say `input`) might contain null cells. We may remove all rows with NULL cells in column `"A"` by doing `input=input[input["A"].notna()]`
- `to_csv()` puts all the row indices into column 0 in the CSV by default, But `read_csv()` considers column 0 as a column of the DataFrame with data of similar datatype when importing.

  This can be resolved by using the keyword argument `index_col` of `read_csv()`. `index_col = False` will take row indices 0,1,2... and `index_col =` an integer will take the column corresponding to that integer as the index column.
- To append elements to a numpy array as it is, `arr=np.append(arr, arr2, axis=0)`. Here `arr2` will get appended as it is, and can also be nested or a single element.
- `a = np.array([]).reshape()` will initialize an empty array. For eg, `np.array([])` gives an empty linear array, `np.array([]).reshape((0,2))` gives an array of duplet arrays.

Machine Learning

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Supervised Learning Algorithms

Supervised learning is when a 'Training Set' of data is first given to the machine for it to gain experience. The machine is used on 'Testing Set' of data only after training it with data that has been pre-labeled manually.

- Classification: The machine must predict the most probable category, class, or label for new examples.
  eg, Decision Tree Classification
- Regression: The machine must predict the value of a continuous response variable. In a graphical visualization, this will be fitting a curve into the given data.
  eg, Linear Regression

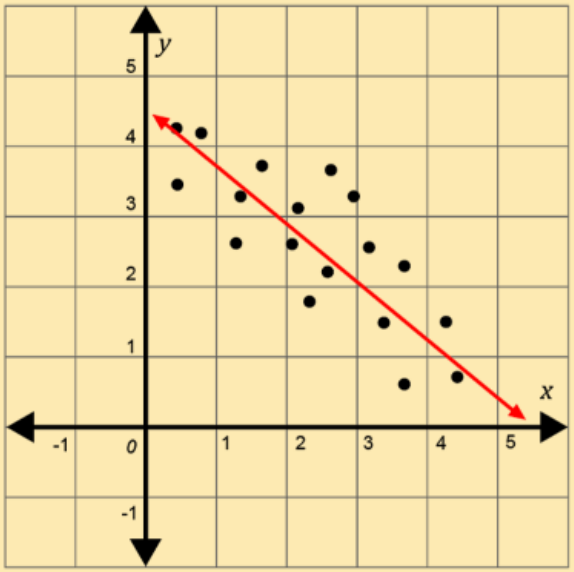
Unsupervised Learning

The machine is directly given the testing set, which is unlabelled data, the Machine gains experience from the same data set and performs its function on it.

- Clustering: Say there are data points given over the Cartesian plane belonging to multiple phenomenon. This could be run through a clustering algorithm to classify them into separate classes.
  eg, K-Means Clustering
- Association: Analysis and prediction of a few events occurring together in correlation.
  eg, Apriori Algorithm

Linear Regression

Linear Regression is the simplest Supervised Learning Learning Algorithm. The Machine learns from a data set that consists of tuples of values. This is stored as a set of coefficients so that the Machine predicts the future outcome using a linear equation with these coefficients.



Line of Best Fit

For some visual intuition, see the figure to the left. Visually, Linear Regression is simply fitting a straight line to the given data.

Mathematically, this is done by minimizing the variance of the predicted values from the training set's values. That is, if y_i is the i^{th} dependent value in the training set and x_i is the i^{th} value of independent variable, $y(x) = a_0 + a_1 x$ is the predicted value, then we choose $\{a_0, a_1, ...\}$ so as to minimize the below quantity:

$$n \cdot \sigma^2 = \sum_{i=0}^{n-1} (y_i - a_0 - a_1 x_i)^2$$

For a minima, we have $\frac{\partial n \cdot \sigma^2}{\partial a_i} = 0$; $\forall$ valid i. Proceeding further, the above equations may be arranged in a matrix as shown below and solved:

$$\begin{bmatrix} \sum 1 & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum y_i x_i \end{bmatrix}$$

Linear Regression using SciKit

Preparing Data:

i) Incase any data is Binary in nature, encode it into 0s and 1s.

```
from sklearn.preprocessing import LabelEncoder
labelencoder_Y = LabelEncoder()
Y = labelencoder_Y.fit_transform(Y)
```

ii) Receive the data in a DataFrame. Remove all rows with NULL Cells.

iii) Get the data into two numpy arrays for X and Y

iv) Split the data into Training and Testing Sets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
↪  test_size = 0.25, random_state = 0)
```

v) Normalize the array if necessary.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Finishing it up:

Until here the procedure is shared between more classification and regression algorithms. Here we may choose whichever algorithm we want to implement and just use the SciKit function for it appropriately.

```
from sklearn.linear_model import LinearRegression

Regression_Object = LinearRegression()

Regression_Object.fit(X_train, Y_train)

Y_pred = Regression_Object.predict(X_test)
```

Here Y_pred would be an `ndarray` of predicted values. This can be plotted against Y_test to compare. Furthermore, we may analyse our fit further by calculating mean square error, score, confusion matrix, accurate and so on.

Practice Tasks 1:

1. For the given data in the CSV file here, Implement Linear Regression using SciKit. Plot the Training data against the Linear Equation and check if the result is as expected.

2. For the same data with $Y > 30$, Implement Linear Regression from Scratch using only Numpy and Pandas. You might want to read upon and use Numpy's `np.linalg.solve()` to solve the matrix equation and `np.sum()` to do the summations within the matrices.

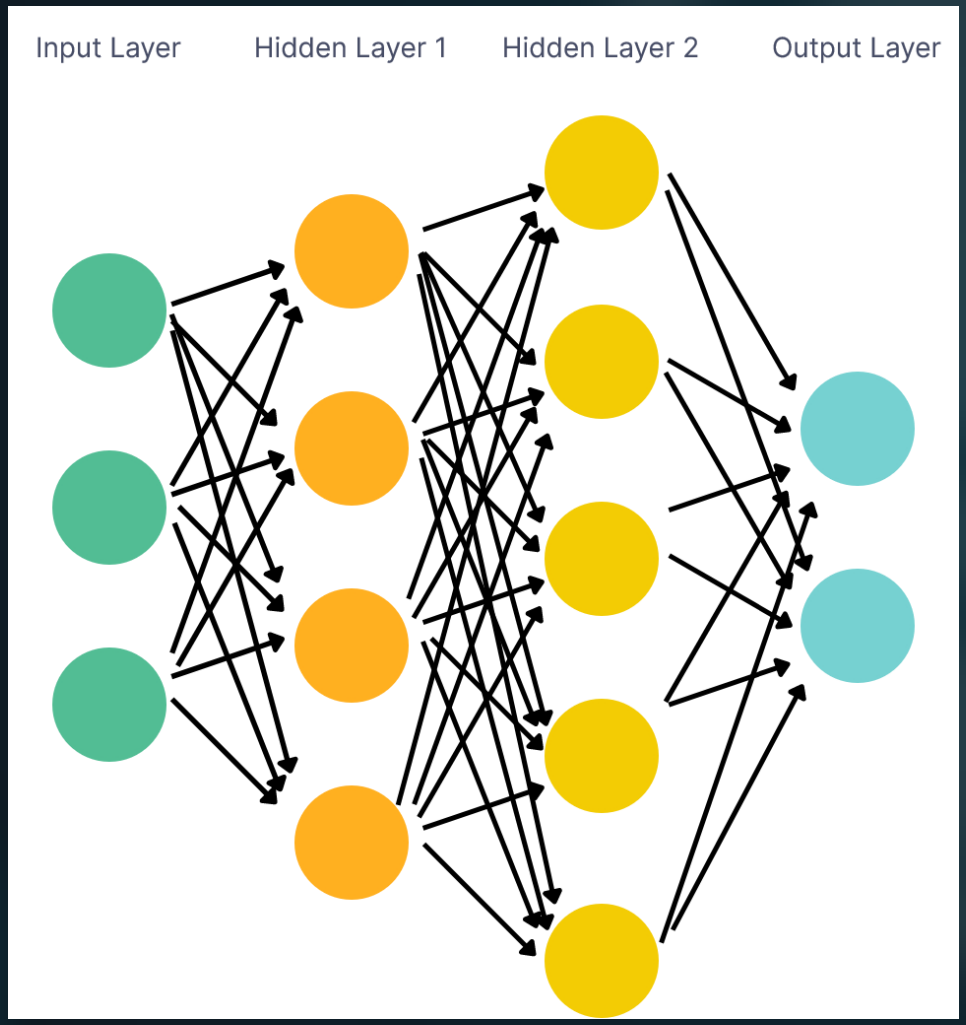
3. (Challenge) Try fitting a Saturation Growth Curve of general form $y = \frac{Ax}{B+x}$ to the same data using SciKit's Linear Regression (Do not use SciKit's Logistic Regression).

4. (Challenge) Following the same procedure as the motivation behind Linear Regression, fit a quadratic polynomial for same data from scratch using only Numpy and Pandas.

Solutions for Tasks 1:

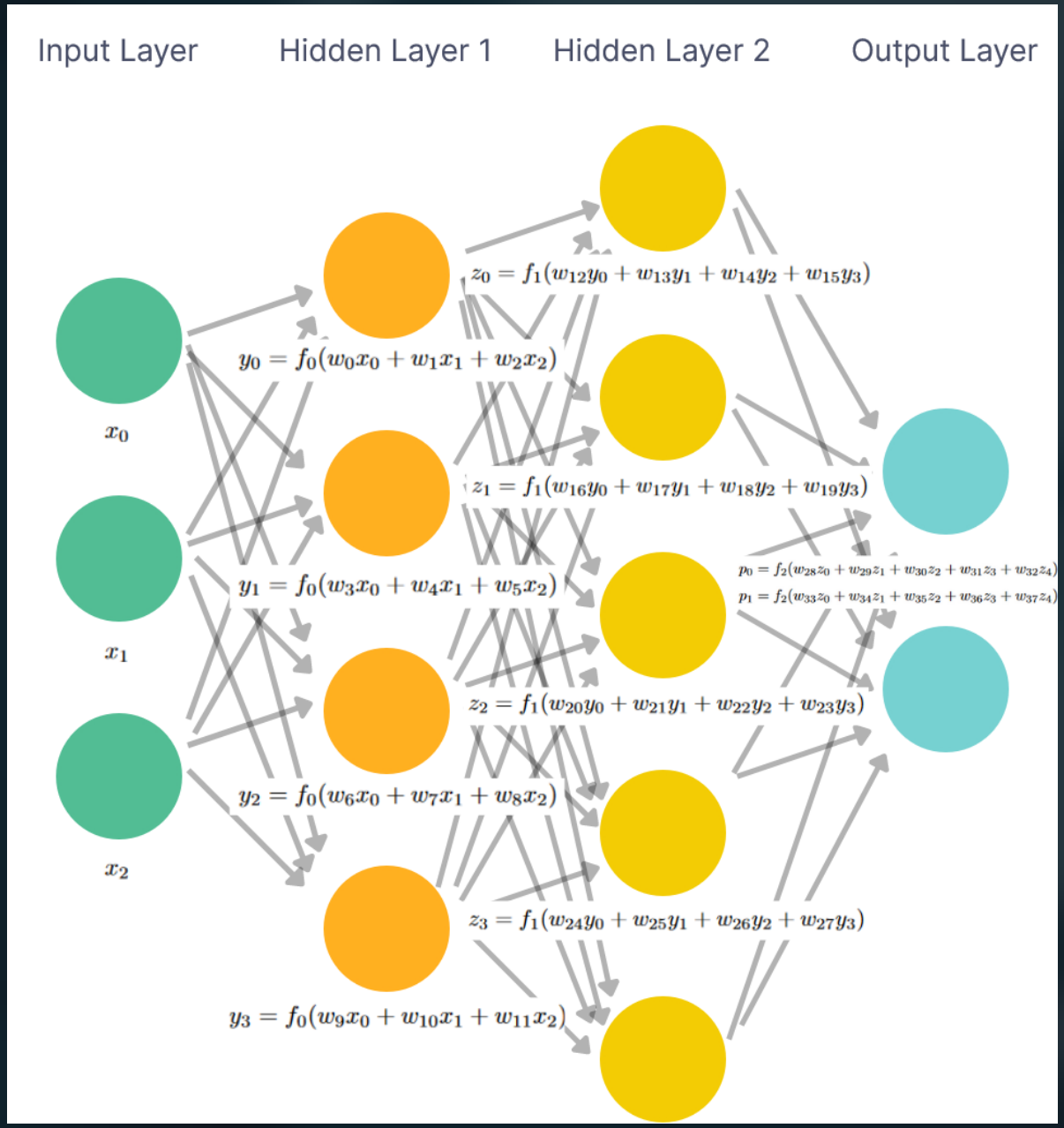
Solutions for Tasks 1:

Neural Networks



- The type of data depends on purpose but it should be vectorizable, each element to one input neuron (can be nested, aka tensor, See here).
- Each arrow takes the output of a neuron to the input of the next neuron, multiplied by a specific weight.
- Each hidden layer neuron transforms the sum of all inputs given. Type of transformation depends on the type of layer.
- Each output neuron is a class and the value that neuron reflects will be the probability that the given input belongs to that class.

Essentially this:



Learning:

The net starts off with an arbitrary set of weights. On each pass of a data entry, the error in the prediction made by the network is calculated. For example, one way of calculating error is the Mean Square Error. If the i^{th} class is the correct one, p_k are probabilities predicted by the network, and $\mathbf{w} = w_1\mathbf{e_1} + w_2\mathbf{e_2}...$, then error is given by

$$E(\mathbf{w}) = (1 - p_i)^2 + \sum_{k \neq i} p_k^2$$

Optimizing the system on each step is done by changing the weights in order to decrease the error. For example, one of the ways to do so is

$$\mathbf{w} \text{ -} = lr \cdot \nabla E$$

Where lr is the "learning rate" and ∇ is the gradient with respect to $\mathbf{w}$

Little more technical details:

- The transformation that a neuron layer does to it's inputs at each neuron is called the "Activation Function of that layer". Let x_k be the sum of inputs the k^{th} neuron.Some common activation functions in Artificial Neural Networks (ANN) are:
  - `'relu'`: $f(x_i) = max(0, x_i)$
  - `'softmax'`: $f(x_i) = \frac{e^{x_i}}{\sum_k e^{x_k}}$
  - `'heaviside'`: $f(x_i) = \frac{sgn(x_i)+1}{2}$
  - `'sigmoid'`: $f(x_i) = \frac{e^x}{1+e^x}$

  If the input is an array, then the output is an array with activation function applied to all elements in the input array.
- The error function example used before is called `keras.losses.MeanSquaredError()`
- The optimization method example used before is called `'sgd'`
- There is also a constant term at each layer. For eg, $y_0 = f_0(w_0x_0 + w_1x_1 + c_0)$, and c_0 will also be a part of the weights vector $\mathbf{w}$
- We omit detailed discussions on over-fitting & under-fitting, read more here if interested.

Coding an ANN with TensorFlow

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

1. Vectorize Data: Do some preprocessing to the data to make it a vector. For example, if a data point is a point in the cartesian plane, then the vector becomes $\begin{bmatrix} x & y \end{bmatrix}^T$ and the input layer will have 2 Neurons. Similarly if the data is black and white image, it could be vectorized by taking the brightness at each pixel and arranging in a matrix.

2. Build the model in `keras`: There are several settings, but for getting started we'll stick to `Sequential` and `Dense`. We won't be discussing about choosing the right size and depth of our network to not over/under-fit for now.

```
model = keras.Sequential(list_of_layer_objects)
```

Let `a` be the number of neurons and `'relu'` the activation, then this layer object would be the following. Note that the argument `input_shape=(4,)` specifies that the input layer has 4 neuron and is included only in the first hidden layer.

```
layers.Dense(a, activation="relu", name="name",
↪  input_shape=(4,))
```

Now we compile the model defined above.

```
model.compile(loss=tf.keras.losses.MeanSquaredError(),
↪  optimizer='adam', metrics=['accuracy'])
```

We use `'adam'` which is a method very close to `'sgd'` since SGD tends to saturate. We may also specify learning rate if instead of `'adam'`, we use `keras.optimizers.Adam(learning_rate=0.001)` .

3. Train it:

```
model.fit(x, y, epochs=150, batch_size=10)
```

- `x`: A numpy array of subarrays of the input shape or a list of numpy arrays of the given number of input neurons.
- `y`: A numpy array of subarrays of the output shape or a list of numpy arrays of the given number of output neurons.
- epochs: An epoch is an iteration over the entire x and y data provided
- batch_size: Number of samples per gradient update

4. Show-time: We may do any of the two below.

```
y_pred = model.predict(x_test) # An Array of predictions
model.evaluate(x_test, y_test) # Performance of the model
```

Where x_test, y_test and y_pred here are parallels to x and y in the `fit()` function.

5. Tuning the Model:

   Experiment with the size of layers, depth of model, learning rate, or batch size of the fit function to enhance our model's performance. We take this extra feedback loop step to fine-tune our model. Here is an example list of tuning that worked with some models, note here that these may not work for all models.

   - If the accuracy saturates, either the optimizer or loss function is unfit or the learning rate is too high.
   - If the accuracy is too low or loss too high, it's a case of under-fit. Increase layer size or depth or decrease batch size when this happens. (or increase data-set size)
   - If accuracy ever drops after a certain epoch or if training accuracy is very high but testing accuracy is low, it's a case of over-fit. Decrease depth or increase batch size.
   - If model's accuracy improves very gradually or only starts increasing after many epochs, increase learning rate or decrease depth or layer size.

Practice Tasks 2:

1. Write a python program along with numpy, random and pandas to generate a CSV dataset with columns `x`,`y` and `label`. Here (x, y) is a point in the unit square $[0,1] \times [0,1]$ and $label = \begin{cases} 1 & \text{if } r \leq \sin 2\theta \\ 0 & \text{otherwise} \end{cases}$

   Build a neural network that predicts if the point (x, y) lies within the region $r < \sin 2\theta$ given x and y using Keras. Use the data set generated by splitting it into training and test sets at a reasonable ratio. Aim for an accuracy more than 0.85 by tuning the neural network.

2. Build an MLP which takes two input either 0/1 and gives XOR output.

If your model for P1 behaves roughly like this (or better), then you are on the right track

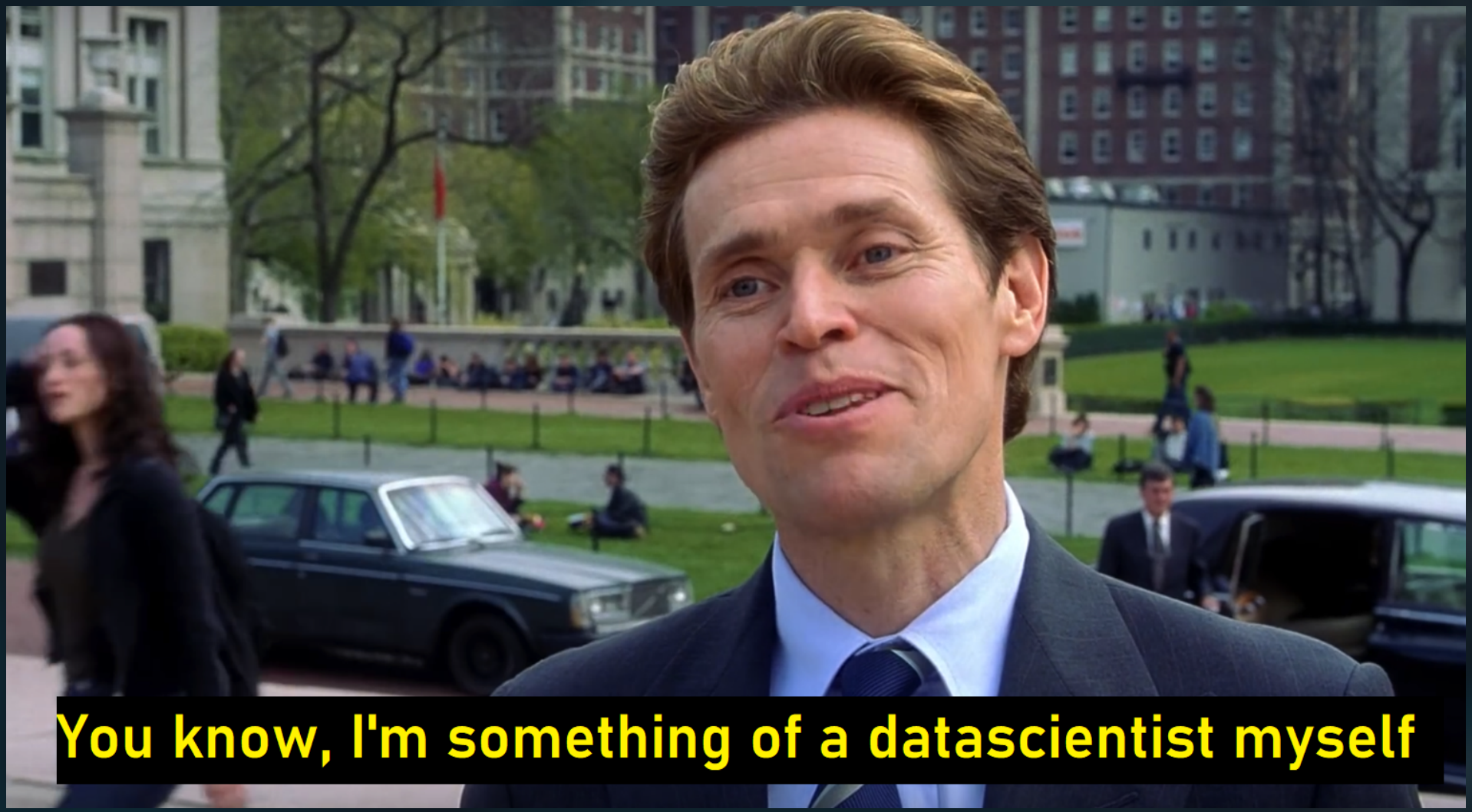
```
Fitting Model:

Epoch 1/10
375/375 [==============================] - 2s 5ms/step - loss: 0.2417 - accuracy: 0.6015
Epoch 2/10
375/375 [==============================] - 2s 5ms/step - loss: 0.2361 - accuracy: 0.6015
Epoch 3/10
375/375 [==============================] - 2s 5ms/step - loss: 0.2025 - accuracy: 0.7030
Epoch 4/10
375/375 [==============================] - 2s 5ms/step - loss: 0.1691 - accuracy: 0.7725
Epoch 5/10
375/375 [==============================] - 2s 5ms/step - loss: 0.1586 - accuracy: 0.7735
Epoch 6/10
375/375 [==============================] - 2s 4ms/step - loss: 0.1533 - accuracy: 0.7785
Epoch 7/10
375/375 [==============================] - 2s 4ms/step - loss: 0.1438 - accuracy: 0.7960
Epoch 8/10
375/375 [==============================] - 1s 3ms/step - loss: 0.1292 - accuracy: 0.8221
Epoch 9/10
375/375 [==============================] - 2s 4ms/step - loss: 0.1105 - accuracy: 0.8487
Epoch 10/10
375/375 [==============================] - 2s 4ms/step - loss: 0.0894 - accuracy: 0.8844


Showtime:

782/782 [==============================] - 2s 2ms/step - loss: 0.0801 - accuracy: 0.8978
```

Summer School Kids after building their first Neural Network be like:

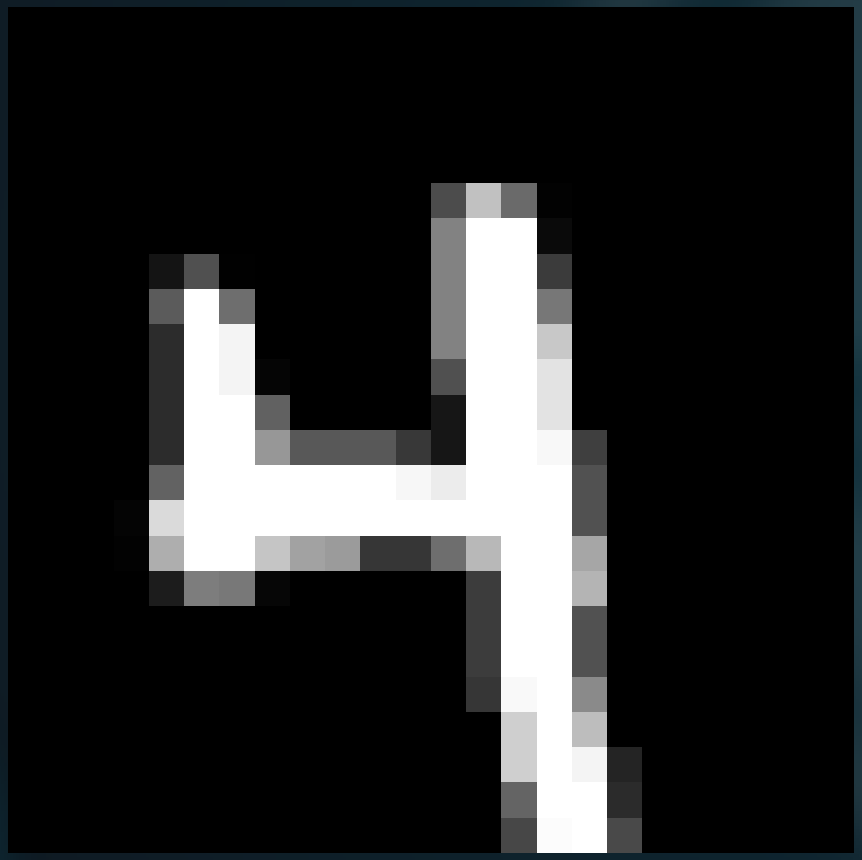


Convolution Neural Networks

We are finally walking into the realms of image processing. Our end goal in this Programming Module is to build a Convolution Neural Network that would tell us what type of waste a given image contains. This would complete our Case Study of the Automatic Waste Segregator project.

Convolution:

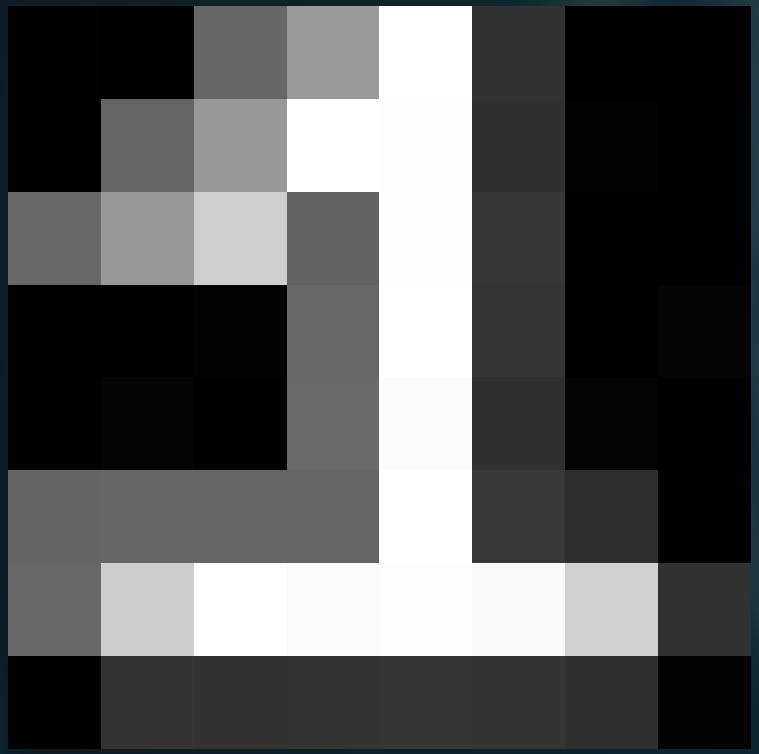
Imagine if you could take an image like this:



And isolate the vertical edges alone like this:

Such transformations on images can be achieved by "Convolution".

An image is vectorized to be fed into a neural network by making it a matrix of vectors, each vector may be of shape (1,) if grayscale, (3,) if RGB or (4,) if the source supports transparency too. Each element in a vector corresponds to the brightness of that colour in that pixel.



$$=\begin{bmatrix} 0 & 0 & 0.4 & 0.6 & 1 & 0.2 & 0 & 0 \\ 0 & 0.4 & 0.6 & 1 & 1 & 0.2 & 0 & 0 \\ 0.4 & 0.6 & 0.8 & 0.6 & 1 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 1 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 1 & 0.2 & 0 & 0 \\ 0.4 & 0.4 & 0.4 & 0.4 & 1 & 0.2 & 0.2 & 0 \\ 0.4 & 0.8 & 1 & 1 & 1 & 1 & 0.8 & 0.2 \\ 0 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0 \end{bmatrix}$$

A "filter" is then "convoluted" with this image. A filter is a smaller matrix. The filter used in the example we started with was $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ for example.

"Convolution" is essentially superposing this filter onto Img.

Say we are performing the convolution $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 2 & 1 \\ 3 & 1 & 2 \end{bmatrix} * \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$

So the superposition would look like this:

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -2 & 2 & 0 \\ 0 & -2 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdots + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 2 \\ 0 & 0 & -2 & 2 \end{bmatrix}$$

Notice that the outermost elements in the resulting 4×4 matrix have contributions that aren't from all four sides. Hence we chuck 'em off. We can just simplify this, but there's a much more efficient way to do the same operation. We omit the mathematical proof for this here.

1. Take transpose of filter, let's call it F^T

2. Sweep the filter through Image matrix, let's call it Img

3. At each position, take the sum of products of the two terms that are at the same spot.

Below is an image to demonstrate a filter that's been swept through the entire image and at the final right bottom position along with the value being calculated.

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 2 & 1 \\ 3 & 1 & 2 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 2 & \{(1\cdot 2)+(-1\cdot 1)+(1\cdot 1)+(-1\cdot 2)\} \end{bmatrix}$$

Mathematically, if $Out = Img * F$, then for valid a, b, i, j we have:

$$Out_{(a,b)} = \sum_{i,j} \left(Img_{(a+i-1,b+j-1)} \times F^{T}_{(i,j)} \right)$$

A Convolution Layer:

Here each neuron has it's own filter and convolves it to the sum of inputs given to that neuron to generate the output of that neuron.

This filter isn't specified by the user but are initialized arbitrarily with given shape ("Kernel Size") by the machine and the elements of this filter are learnt just the way weights are learnt from the data-set.

Note that when we vectorized the image, our array shape was $(x, y, 3)$ and not $(3, x, y)$. This might seem like the first convolution layer is taking input from an input layer with 1 neuron, but the 3 channels will infact behave like 3 having input neurons here. In short,

$$\boxed{\text{no. of input neurons} = \text{no. of channels in the 1st convolution layer}}$$

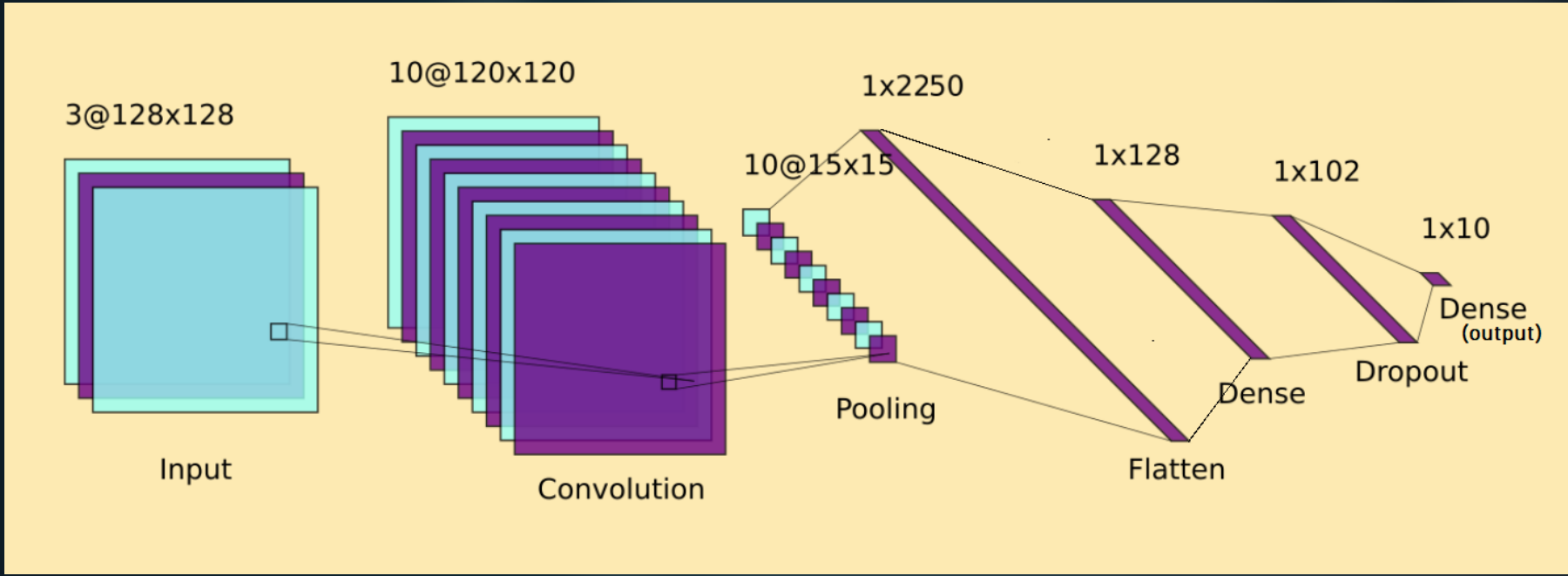
Other Layers in a CNN:

- Pooling Layer: Pooling is essentially decreasing the resolution of the image. Starting from left top, this layer will take a group of elements of a given shape ("Pool size") and puts one element in the output corresponding to this group. For example, a max-pool with pool size $(2,2)$ on a matrix will take the maximum element in each 2×2 group like shown below.

$$\begin{bmatrix} 9 & 4 & 2 & 1 \\ 2 & 1 & 5 & 4 \\ 3 & 1 & 5 & 7 \\ 1 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 9 & 5 \\ 3 & 7 \end{bmatrix}$$

  Note that unlike Convolution where the filter was swept through every position possible, Pooling is done without repeating elements in groups made.
- Flatten Layer: Reshapes image matrices in the previous layer to make each pixel a neuron.
- Dropout Layer: Disables a few neurons & normalize the rest to make up for it in the previous layer to avoid over-fitting.

In the end our CNN would look like this:



Implementing CNN

Implementation follows the same outline as an ANN. Special attention required only for the vectorizing of data and the syntaxes for the layer objects for the layers that are used in a CNN.

Above we've covered the background for all of these tasks already, we just need to put them in place:

1. Vectorizing an Image:

```
Img = plt.imread(r"image.jpg") # RGB
Img = plt.imread(r"image.jpg",0) # Grayscale
```

This will give a numpy array of the right shape by itself.

2. Visualizing an Image Matrix:

```
plt.Imshow(image) # RGB
plt.show()
```

These two will take an array of the right shape and display the image.

```
plt.Imshow(image, cmap='Greys') # Grayscale
plt.show()
```

3. Layer Objects:

- Convolution Layer: The input_shape argument is included only when this layer is the first hidden layer.

```
layers.Conv2D(10, kernel_size=(3,3), input_shape=(128,128,3),
↪  activation='relu')
```

- Flatten Layer:

```
layers.Flatten()
```

- Dropout Later: Let 0.2 be the rate of disabling.

```
layers.Dropout(0.2)
```

- Max-Pooling Layer:

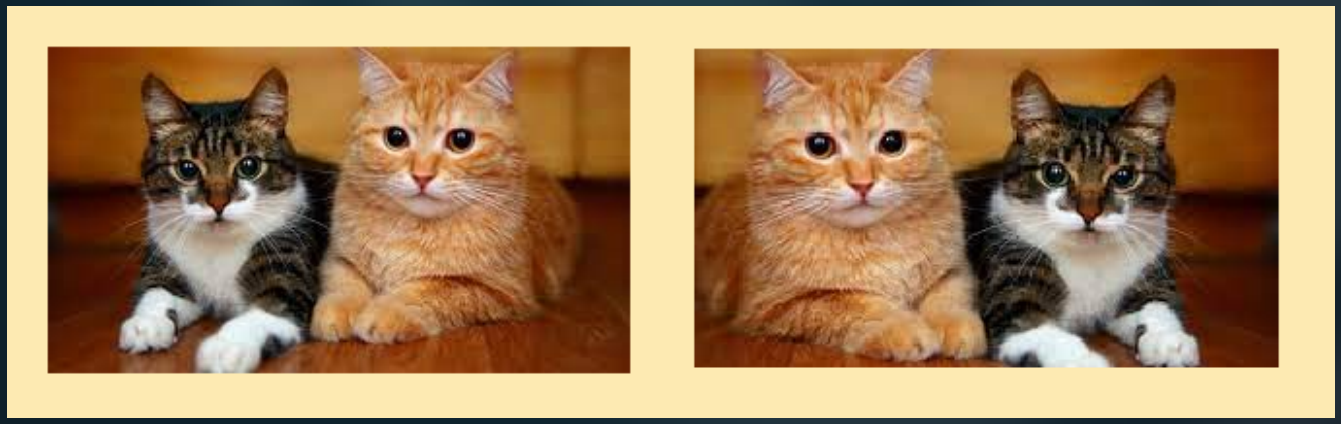
```
layers.MaxPooling2D(pool_size=(4, 4))
```

4. ImageDataGenerator

"Being able to make the most out of very little data is a key skill of a competent data scientist"

This practice battles three different issues:

(a) Small Data-Set: On small data sets, it's hard to achieve anything above 60% usually. But notice these two images below. To us, these are the same, but to a computer these are entirely different.



This way, ImageDataGenerator extracts the most out of every image.

(b) Overfitting: The same image isnt fed into the model every epoch. Thus it won't overfit the same image.

(c) Memory: If the image data is stored as a numpy array instead of a generator object like this, it will fill out GPU and RAM very very quick. About 2k images at 150 × 150 takes more than 2GB's memory.

Using a the `flow_from_directory()` attribute of the generator will let image data flow directly from where it is stored at each batch of data instead of storing an entire epoch in RAM.

Arrange the training directory in form `<training_dir>/class/images.jpg`. Then replace the usual code in the image-vectorizing part and the training part with this following code:

```
from keras.preprocessing.image import ImageDataGenerator
# Generator Object with transformation settings
train_datagen = ImageDataGenerator(
        rescale=1./255,      shear_range=0.2,
        zoom_range=0.2,      horizontal_flip=True,
        vertical_flip=True, width_shift_range=0.2,
        height_shift_range=0.2)
# Vectorize Images in Training Directory
train_generator = train_datagen.flow_from_directory(
        './train',           target_size=(192, 256),
        batch_size=32,       class_mode='categorical')

#Similarly generate validation set too called
↪  validation_generator

# Train Model
model.fit_generator(
        train_generator, steps_per_epoch=2000,
        epochs=50, validation_data=validation_generator,
        validation_steps=800)
```

Practice Tasks 3:

1. Code a Convolution Neural Network to categorize handwritten digits given as an image of resolution (,) and in grayscale.

   Obtain the data set by using the below code: (You won't need ImageDataGenerator for this problem)

```
(x_train, y_train), (x_test, y_test) =
↪  tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

Final Steps:

Lastly, any software we write for solving a social problem will have to be deployed. We will be having specifications to meet like hardware limitations on the micro-controller, minimum workable accuracy, maximum processing time, etc.

Some possible changes could be:

- Changing image resolution for improving processing speed or meeting hardware limits. (`cv2.resize(img, (1280,720))`
- Fine-tuning the model even more.
- Adding a time-delay in the Micro-controller or a clock to synchronise to.

Summing it all up:

Here is a CNN Implemented for Waste Segregation.

Thank You!

Programming Module by,
Aditya Mohapatra
Rishi Nandha V
Tejash Aarav