Project Flow

Farmers can interact with the portal build:

- The main aim of our project is to built a bridge of communication between the farmers and customers across the country so that they can get communicate together and can talk about any product related queries for both ends. The main task will be a challenge to the most of the farmers because they are lacking the knowledge about the new technology and trends which is used in this fast developing world. The main success of our project is to provide the fruitful benefits for both the customer as well as to the farmers, providing the knowledge and covering different aspects of the resources that they are unaware of till date.

- The main objective of this project is that there is an direct communication is done in between the User and the farmer.

- Also farmer can sell the product direct to the customer and the profit will get to the farmer.

- Also the weather information is get to the farmer, in which weather did farmer should grow the particular crop.

- Also user can communicate for buying the crop or order the particular crop User can communicate through “Chat “.

Interacts with the user interface to upload images of diseased:

- This paper presents a mobile-based system for detecting plant leaf diseases using Deep Learning (DL) in realtime. In particular, we developed a distributed system that is organized with parts executing on centralized servers on the cloud and locally on the user’s mobile devices. We created a dataset that consists of more than 96 k images for the most common 38 plant disease categories in 14 crop species, including apple scab, apple black rot, cherry powdery mildew, corn common rust, grape leaf blight, and many others.

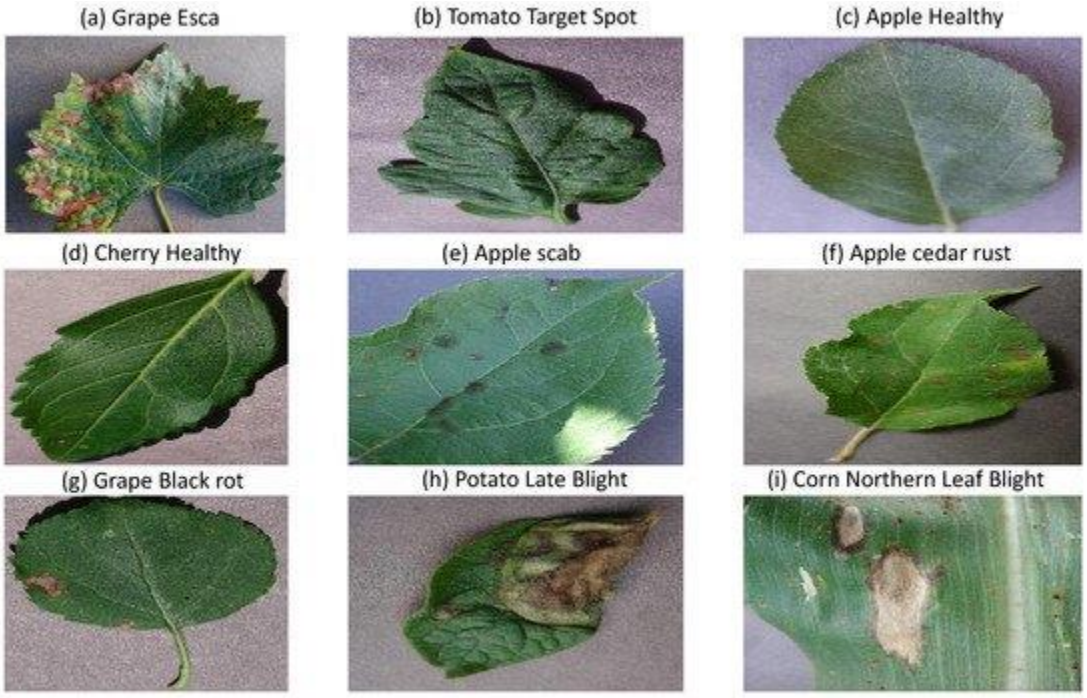


Figure 1. Samples from our Imagery Dataset that Show Different Types of Healthy and Diseased Plant Leaves.

- At the cloud side, we created a Convolutional Neural Network (CNN) model [12] that can feed images directly from farmers' mobile devices. The model then performs object detection and semantic segmentation, and displays the disease category along with the confidence percentage and classification time have taken to process the image. We developed an Android mobile app to allow limited-resources farmers to capture a photo of the diseased plant leaves.

- The mobile app runs on top of the CNN model on the user side. Also, the application displays the confidence percentage and classification time taken to process the image.

Our model built analyses the Disease and suggests the farmer with fertilizers are to be used :

- The goal of research on fertilizer rate is to determine the amount of fertilizer needed to achieve a commercial crop yield with sufficient quality that is economically acceptable for the grower. In Florida, these types of studies take a slightly different approach

depending on whether soil testing for the nutrient in question is involved. For example, rate studies with nitrogen (N) on sandy soils would not involve soil testing, but rate studies with phosphorus (P) or potassium (K) would.

- In the case of N on sandy soils, the researcher assumes there is minimal N supplied from the soil that would satisfy the crop nutrient requirement. In the case of P or K, a properly calibrated soil test will reveal if a response (yield and fruit quality) to the nutrient is likely or not. Rate studies are best conducted on soils low in the particular nutrient so that maximum crop response is likely and that response can be modeled.

- Proper experimental design and statistical data analyses are critical to interpretation of the results. Research begins with a hypothesis or a set of hypotheses. One possible hypothesis may be that there will be no effect on yield associated with N fertilization.

- This hypothesis, called the null hypothesis, is evaluated with an experiment to test crop yield response against a range of N rates in a field likely to produce a large response to the addition of N fertilizer.

- The researcher applies a range of fertilizer rates thought to capture the likely extent of possible crop yield responses. A zero-fertilizer treatment is **always** included. Crop response without an actual fertilizer application demonstrates and measures the soil-supplied effects, if any.

- In some cases, sufficient nutrients, or at least a low portion of the crop nutrient requirement, may come from the soil, while in other cases, nutrients may come from the irrigation water.

- The researcher may decide to divide the total seasonal amount of fertilizer into split-applications, following what would likely be a recommended practice for the crop being studied. Multiple applications avoid potential large losses of fertilizer because of rainfall events, especially for nutrients that are mobile in the soil.

- Typically, all treatment rates are handled similarly for timing and placement of the fertilizer to minimize any confounding effects with rate.

- During the growing season, the researcher may sample the plant for nutrient concentrations, using whole dried leaves and/or fresh petiole sap. These samples will help the researcher prove the response in yield was related to the plant's nutrient status.

- Typically, soil samples are not used because there is a chance of including a fertilizer particle in the sample, or there may be questions of where to sample if the fertilizer is applied by banding or through a drip tape. Photographs taken during the season are useful for documenting both growth and potential plant deficiency symptoms.

- The crop response of interest, typically marketable yield, is measured at the appropriate harvest time(s). For vegetables, the fruits are evaluated according to USDA grade standards to detect any effects of fertilization on fruit quality (size, color, sugar content, etc.).

- Yields are expressed in the prevailing commercial units per area of production (e.g., 28-lb boxes/acre, 42-lb crates/acre, bushels/acre, tons/acre, etc.). The raw data should be plotted in a scatter diagram (Figure 1) to gain insight into the type and magnitude of response. Plotting the raw data allows the researcher to inspect for apparent atypical data points that may illustrate errors somewhere in the data entry process.

Classify the dataset into train and test sets:

- Training to the test set is a type of overfitting where a model is prepared that intentionally achieves good performance on a given test set at the expense of increased generalization error.

- It is a type of overfitting that is common in machine learning competitions where a complete training dataset is provided and where only the input portion of a test set is provided. One approach to **training to the test set** involves constructing a training set that most resembles the test set and then using it as the basis for training a model.

- The model is expected to have better performance on the test set, but most likely worse performance on the training dataset and on any new data in the future.

- Although overfitting the test set is not desirable, it can be interesting to explore as a thought experiment and provide more insight into both machine learning competitions and avoiding overfitting generally.

- In this tutorial, you will discover how to intentionally train to the test set for classification and regression problems.

After completing this tutorial, you will know:

- Training to the test set is a type of data leakage that may occur in machine learning competitions.
- One approach to training to the test set involves creating a training dataset that is most similar to a provided test set.
- How to use a KNN model to construct a training dataset and train to the test set with a real dataset.

This tutorial is divided into three parts; they are:

1. Train to the Test Set.
2. Train to Test Set for Classification.
3. Train to Test Set for Regression.

1.Train to the Test Set:

- In applied machine learning, we seek a model that learns the relationship between the input and output variables using the training dataset.
- The hope and goal is that we learn a relationship that generalizes to new examples beyond the training dataset. This goal motivates why we use resampling techniques like k-fold cross-validation to estimate the performance of the model when making predictions on data not used during training.
- In the case of machine learning competitions, like those on Kaggle, we are given access to the complete training dataset and the inputs of the test dataset and are required to make predictions for the test dataset.
- This leads to a possible situation where we may accidentally or choose to train a model to the test set. That is, tune the model behavior to achieve the best performance on the test dataset rather than develop a model that performs well in general, using a technique like k-fold cross-validation.

Training to the test set is often a bad idea:

- It is an explicit type of data leakage. Nevertheless, it is an interesting thought experiment.

- One approach to training to the test set is to contrive a training dataset that is most similar to the test set.

- For example, we could discard all rows in the training set that are too different from the test set and only train on those rows in the training set that are maximally similar to rows in the test set.
- We can use a k-nearest neighbor model to select those instances of the training set that are most similar to the test set. The KNeighborsRegressor and KNeighborsClassifier both provide the kneighbors() function that will return indexes into the training dataset for rows that are most similar to a given data, such as a test set.

- We might want to try removing duplicates from the selected row indexes.

```
1 ...
2 # remove duplicate rows
3 ix = unique(ix)
```

- We can then use those row indexes to construct a custom training dataset and fit a model.

```
1 ...
2 # create a training dataset from selected instances
3 X_train_neigh, y_train_neigh = X_train[ix], y_train[ix]
```

- Given that we are using a KNN model to construct the training set from the test set, we will also use the same type of model to make predictions on the test set. This is not required, but it makes the examples simpler.

- Using this approach, we can now experiment with training to the test set for both classification and regression datasets.

2.Train to Test Set for Classification:

- We will use the diabetes dataset as the basis for exploring training for the test set for classification.
- Program Each record describes the medical details of a female and the prediction is the onset of diabetes within the next five years.
  - ✓ Dataset Details: pima-indians-diabetes.names
  - ✓ Dataset: pima-indians-diabetes.csv
- The dataset has eight input variables and 768 rows of data; the input variables are all numeric and the target has two class labels, e.g. it is a binary classification task.

- First, we can load the dataset directly from the URL, split it into input and output elements, then split the dataset into train and test sets, holding thirty percent back for the test set. We can then evaluate a KNN model with default model hyperparameters by training it on the training set and making predictions on the test set.
- The complete example is listed below.

```
# example of evaluating a knn model on the diabetes classification dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
# load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.csv'
df = read_csv(url, header=None)
data = df.values
X, y = data[:, :-1], data[:, -1]
print(X.shape, y.shape)
# split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# define model
model = KNeighborsClassifier()
# fit model
model.fit(X_train, y_train)
# make predictions
yhat = model.predict(X_test)
# evaluate predictions
accuracy = accuracy_score(y_test, yhat)
```

```
print('Accuracy: %.3f' % (accuracy * 100))
```

- Running the example first loads the dataset and summarizes the number of rows and columns, matching our expectations. The shape of the train and test sets are then reported, showing we have about 230 rows in the test set.

```
(768, 8) (768,)
(537, 8) (231, 8) (537,) (231,)
Accuracy: 77.056
```

- Finally, the classificationaccuracy of the model is reported to be about 77.056 percent.
- Now, let's see if we can achieve better performance on the test set by preparing a model that is trained directly for it.
- First, we will construct a training dataset using the simpler example in the training set for each row in the test set.

```
...
# select examples that are most similar to the test set
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
# get the most similar neighbor for each point in the test set
neighbor_ix = knn.kneighbors(X_test, 1, return_distance=False)
ix = neighbor_ix[:,0]
# create a training dataset from selected instances
X_train_neigh, y_train_neigh = X_train[ix], y_train[ix]
print(X_train_neigh.shape, y_train_neigh.shape)
```

- Next, we will train the model on this new dataset and evaluate it on the test set as we did before.

```
1 ...
2 # define model
3 model = KNeighborsClassifier()
4 # fit model
5 model.fit(X_train_neigh, y_train_neigh)
```

- The complete example is listed below.
- Running the example, we can see that the reported size of the new training dataset is the same size as the test set, as we expected.
- We can see that we have achieved a lift in performance by training to the test set over training the model on the entire training dataset. In this case, we achieved a classification accuracy of about 79.654 percent compared to 77.056 percent when the entire training dataset is used.

```
1 (768, 8) (768,)
2 (537, 8) (231, 8) (537,) (231,)
3 (231, 8) (231,)
4 Accuracy: 79.654
```

- You might want to try selecting different numbers of neighbors from the training set for each example in the test set to see if you can achieve better performance.
- Also, you might want to try keeping unique row indexes in the training set and see if that makes a difference.
- Finally, it might be interesting to hold back a final validation dataset and compare how different "*train-to-the-test-set*" techniques affect performance on the holdout dataset. E.g. see how training to the test set impacts generalization error.

- Report your findings in the comments below.

- Now that we know how to train to the test set for classification, let's look at an example for regression.

3.Train to Test Set for Regression:

- We will use the housing dataset as the basis for exploring training for the test set for regression problems.

- The housing dataset involves the prediction of a house price in thousands of dollars given details of the house and its neighborhood.
  - ✓ Dataset Details: housing.names
  - ✓ Dataset: housing.csv
- It is a regression problem, meaning we are predicting a numerical value. There are 506 observations with 13 input variables and one output variable.

```
1 0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,1,296.0,15.30,396.90,4.98,24.00
2 0.02731,0.00,7.070,0,0.4690,6.4210,78.90,4.9671,2,242.0,17.80,396.90,9.14,21.60
3 0.02729,0.00,7.070,0,0.4690,7.1850,61.10,4.9671,2,242.0,17.80,392.83,4.03,34.70
4 0.03237,0.00,2.180,0,0.4580,6.9980,45.80,6.0622,3,222.0,18.70,394.63,2.94,33.40
5 0.06905,0.00,2.180,0,0.4580,7.1470,54.20,6.0622,3,222.0,18.70,396.90,5.33,36.20
6 ...
```

- A sample of the first five rows is listed below.

- First, we can load the dataset, split it, and evaluate a KNN model on it directly using the entire training dataset. We will report performance on this regression class using mean absolute error (MAE).
- The complete example is listed below.

```
1  # example of evaluating a knn model on the housing regression dataset
2  from pandas import read_csv
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import mean_absolute_error
5  from sklearn.neighbors import KNeighborsRegressor
6  # load the dataset
7  url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv'
8  df = read_csv(url, header=None)
9  data = df.values
```

```
X, y = data[:, :-1], data[:, -1]
print(X.shape, y.shape)
# split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# define model
model = KNeighborsRegressor()
# fit model
model.fit(X_train, y_train)
# make predictions
yhat = model.predict(X_test)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

- Running the example first loads the dataset and summarizes the number of rows and columns, matching our expectations. The shape of the train and test sets are then reported, showing we have about 150 rows in the test set.
- Finally, the MAE of the model is reported to be about 4.488.

```
(506, 13) (506,)
(354, 13) (152, 13) (354,) (152,)
MAE: 4.488
```

- Now, let's see if we can achieve better performance on the test set by preparing a model that is trained to it.
- First, we will construct a training dataset using the simpler example in the training set for each row in the test set.

```
...
# select examples that are most similar to the test set
```

```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
# get the most similar neighbor for each point in the test set
neighbor_ix = knn.kneighbors(X_test, 1, return_distance=False)
ix = neighbor_ix[:,0]
# create a training dataset from selected instances
X_train_neigh, y_train_neigh = X_train[ix], y_train[ix]
print(X_train_neigh.shape, y_train_neigh.shape)
```

- Next, we will train the model on this new dataset and evaluate it on the test set as we did before.

```
...
# define model
model = KNeighborsClassifier()
# fit model
model.fit(X_train_neigh, y_train_neigh)
```

- The complete example is listed below.

```
# example of training to the test set for the housing dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.neighbors import KNeighborsRegressor
# load the dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv'
df = read_csv(url, header=None)
data = df.values
X, y = data[:, :-1], data[:, -1]
```

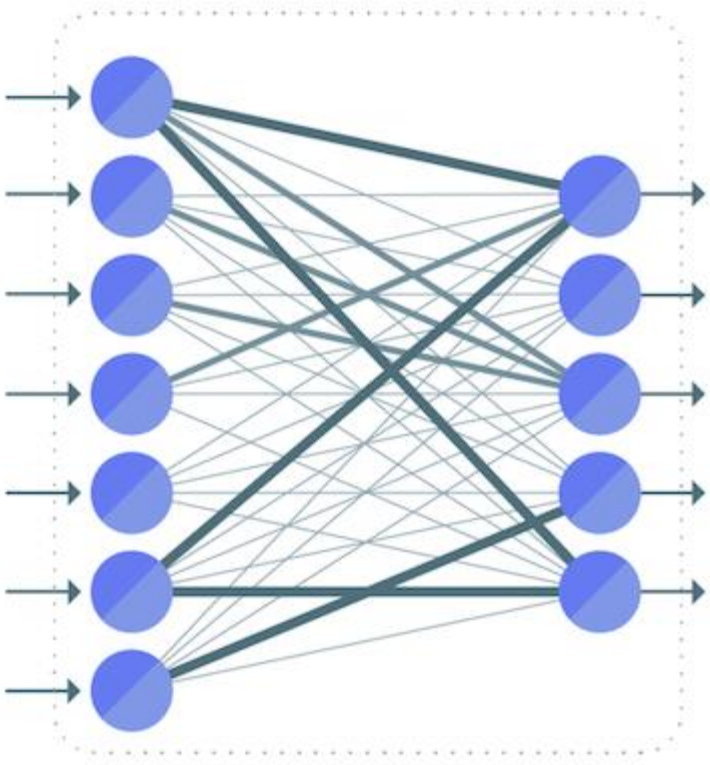
```
print(X.shape, y.shape)
# split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# select examples that are most similar to the test set
knn = KNeighborsRegressor()
knn.fit(X_train, y_train)
# get the most similar neighbor for each point in the test set
neighbor_ix = knn.kneighbors(X_test, 1, return_distance=False)
ix = neighbor_ix[:,0]
# create a training dataset from selected instances
X_train_neigh, y_train_neigh = X_train[ix], y_train[ix]
print(X_train_neigh.shape, y_train_neigh.shape)
# define model
model = KNeighborsRegressor()
# fit model
model.fit(X_train_neigh, y_train_neigh)
# make predictions
yhat = model.predict(X_test)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

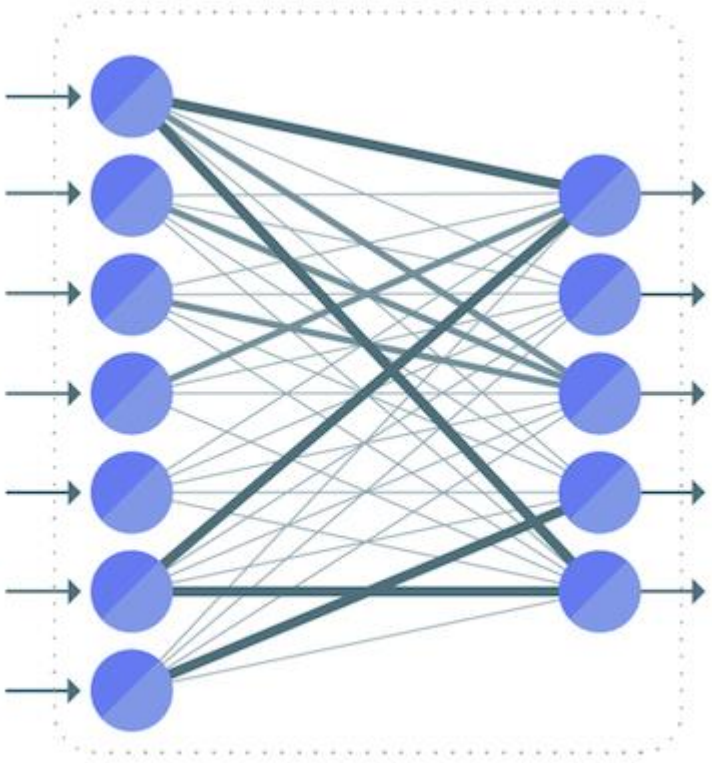
- Running the example, we can see that the reported size of the new training dataset is the same size as the test set, as we expected.
- We can see that we have achieved a lift in performance by training to the test set over training the model on the entire training dataset.

Neural networks Layers:

- Neural networks (NN) are the backbone of many of today's machine learning (ML) models, loosely mimicking the neurons of the human brain to recognize patterns from input data. As a result, numerous types of neural network topologies have been designed over the years, built using different types of neural network layers.
- The Two most common types of neural network layers are *Fully connected*, *Convolution*, and below you will find what they are and how they can be used.

1. Fully Connected Layer:



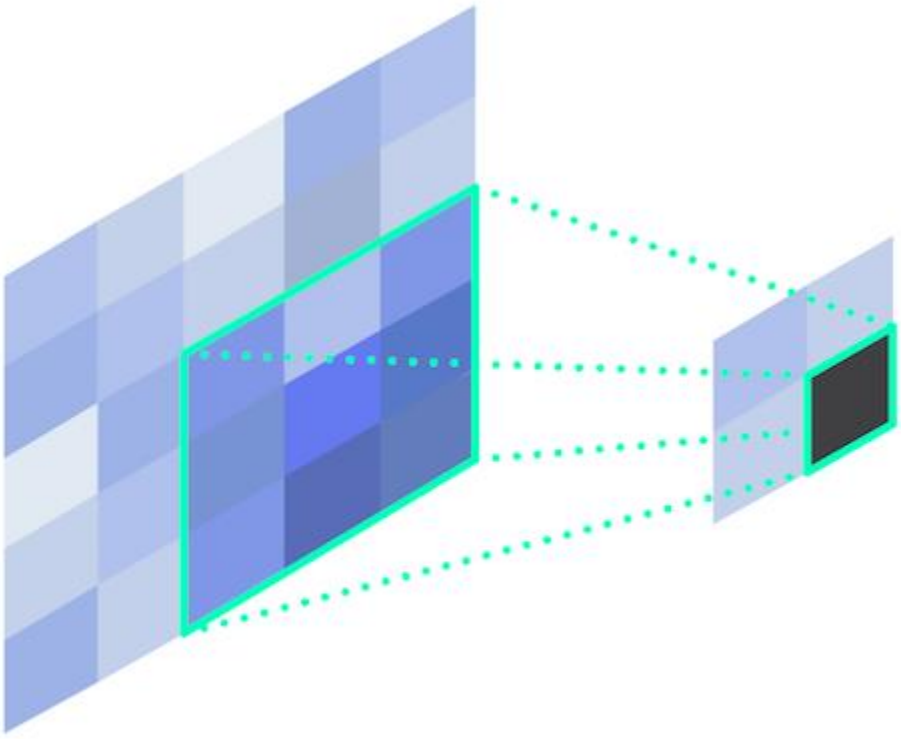


- Fully connected layers connect every neuron in one layer to every neuron in the next layer. Fully connected layers are found in all different types of neural networks ranging from standard neural networks to convolutional neural networks (CNN).

Use Cases:

- Experimentation or learning ML using fully connected neural networks.
- In CNNs to classify images for computer vision.

2.Convolution Layer:



- A Convolution Layer is an important type of layer in a CNN. Its most common use is for detecting features in images, in which it uses a *filter* to scan an image, a few pixels at a time, and outputs a *feature map* that classifies each feature found.

- Multiplication is systematically repeated from left to right and top to bottom over the entire image to detect features. The number of pixels by which the filter moves for the next iteration is called the *stride*.

Use Cases:

- Analyzing imagery for image recognition and classification.

Test The Model:

- The model is to be tested with different images to know if it is working correctly.
- Import the packages and load the saved model
- Import the required libraries.

```
from keras.preprocessing import image
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.models import load_model
import numpy as np
```

- Initially, we will be loading the fruit model. You can test it with the vegetable model in a similar way.

```
model = load_model("fruit.h5")
```

- Load the test image, pre-process it and predict.
- Pre-processing the image includes converting the image to array and resizing according to the model. Give the pre-processed image to the model to know to which class your model belongs to.

```
img = image.load_img('apple_healthy.JPG',target_size = (128,128))
```

```
x  = image.img_to_array(img)
x = np.expand_dims(x,axis = 0)
```

```
pred = model.predict_classes(x)
```

```
pred
```

```
[1]
```

- The predicted class is 1.

Build a Web application using a flask that integrates with the model built:

- Now that we have our model, we will start developing our web application with Flask. Those of you starting out in Flask can read about it here.

- Flask learning resources: Flask Tutorials by Corey Schafer, Learn Flask for Python — Full Tutorial

1. Install Flask:

- You can use the 'pip install flask' command. I use the PyCharm IDE to develop flask applications. ***To easily install libraries in PyCharm*** follow these steps.

2. Import necessary libraries, initialize the flask app, and load our ML model:

- We will initialize our app and then load the "model.pkl" file to the app.

```
#import libraries
 import numpy as np
from flask import Flask, render_template,request
import pickle#Initialize the flask App
app = Flask(__name__)
model = pickle.load(open('model.pkl', 'rb'))
```

3. Define the app route for the default page of the web-app :

- Routes refer to URL patterns of an app (such as myapp.com/home or myapp.com/about). @app.route("/") is a Python decorator that Flask provides to assign URLs in our app to functions easily.

```
#default page of our web-app
@app.route('/')
def home():
 return render_template('index.html')
```

- The decorator is telling our @app that whenever a user visits our app domain (*localhost:5000 for local servers)* at the given .route(), execute the home() function.

4. Redirecting the API to predict the CO2 emission :

- We create a new app route (‘/predict’) that reads the input from our ‘index.html’ form and on clicking the predict button, outputs the result using render_template.

Let’s have a look at our **index.html** file :

```
<div class="login">
   <h1>Predict CO2 Emission of Vehicles</h1>
    <h3> Enter the following values to predict the CO2 emission from the vehicle</h3>

    <!-- Main Input For Receiving Query to our ML -->
   <form action="{{ url_for('predict')}}"method="post">
       <input type="text" name="enginesize" placeholder="Engine Size" required="required" />
       <input type="text" name="cylinders" placeholder="Cylinders" required="required" />
       <input type="text" name="fuel" placeholder="Fuel" required="required" />

       <button type="submit" class="btn">Predict</button>
   </form>

  <br>
  <br>
  {{ prediction_text }}

</div>
```

5. Starting the Flask Server :

```
if __name__ == "__main__":
     app.run(debug=True)
```

- app.run() is called and the web-application is hosted locally on *[localhost:5000].*