Jaineil Mandavia
Project code and dataset: https://github.com/jaineil/Multiclass-Image-Classifier
Email: mandaviajaineil@gmail.com

The inspiration and evolution of this project

Inspired by an episode (S04E04) on the popular comedy TV series, Silicon Valley (HBO), initially I decided to built a desi version of a Binary Image Classifier (similar to the show, which built a Hot Dog or Not a Hot Dog) - Vada Pav or Not a Vada Pav.

I trained a Convolutional Neural Network (CNN) model on a dataset built from images of Vada Pav and Burger, where the output was binary classification of test data (1-Vada Pav / 0-Not A Vada Pav).

Eventually, a layer of abstraction was added to this project for learning Multi-class Image Classification by including 3 classes of images for training the model: Vada Pav, Burger and Sandwiches. And the result entailed correctly labelling the test day as one of the 3 aforementioned categories (Vada Pav, Burger and Sandwiches). This led to the building of a Multi-class (single-label for each class) Image Classifier.

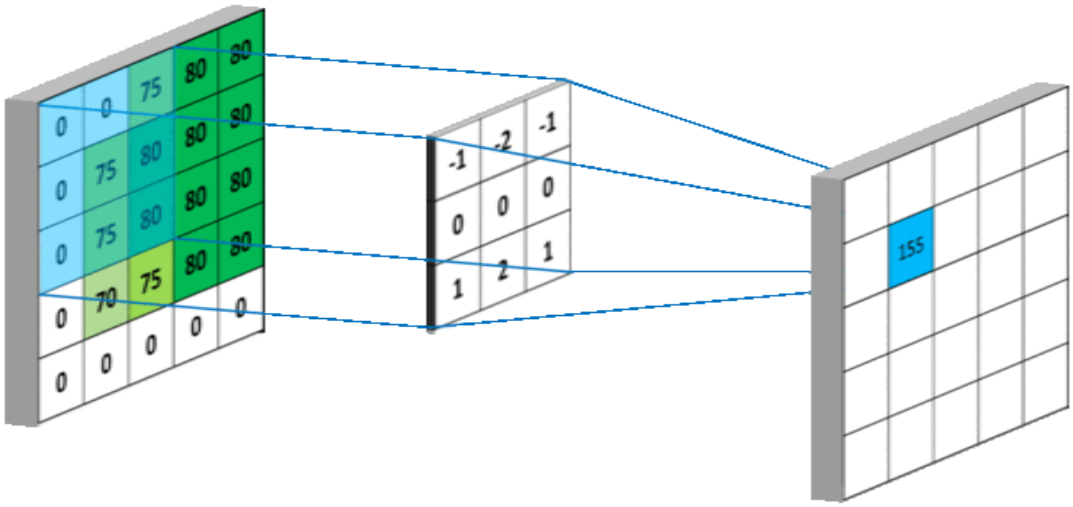
Assumptions

In deep learning, Convolutional Neural Networks (CNN or ConvNet) are complex, feed-forward neural networks. CNNs are used for image classification and recognition because of their high accuracy. The CNN follows a hierarchical model which works on building a network, like a funnel, and finally gives out a fully-connected layer where all the neurons are connected to each other and the output is processed.

There are a lot of algorithms that people used for image classification before CNN became popular. People used to create features from images and then feed those features into some classification algorithm, like SVM. CNNs can be thought of as automatic feature extractors from the image, it does not require feature engineering.

Some algorithms also used the pixel level values of images as a feature vector too. While if I use an algorithm with a pixel vector, I lose a lot of spatial interaction between pixels. A CNN effectively uses adjacent pixel information to effectively

downsample the image first by convolutions & max-pooling, and then uses a prediction layer at the end. This increases the accuracy.



Technologies

I used the Python programming language along with it's popular Deep Learning library - TensorFlow (TFlearn). (This helped me with the convolutions and max-pooling on the images as well as applying the Softmax layer as the last layer of my network!)

I used the NumPy library for Python which added support for large, multi-dimensional arrays and matrices. (Helped me convert the dataset images to NumPy arrays!)

I used the OpenCV library for Python which converted my images to grayscale and resized them before they were labelled and added to the NumPy array.

I used the Matplotlib library for Python which helped me generate an output image showing classification being done on the test data.

Other libraries like tqdm and os were used to add the progress-bar feature and help manage file paths respectively.

```
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt
from random import shuffle
from tqdm import tqdm
import tflearn
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.estimator import regression
```

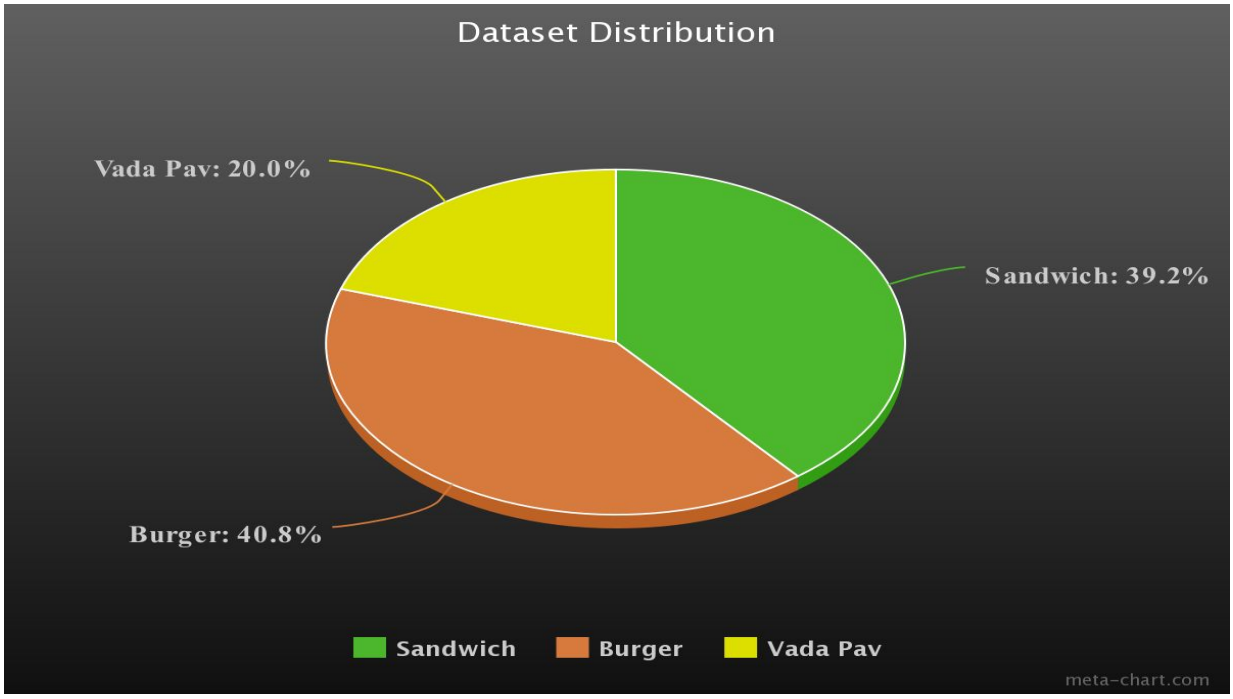
Datasets

https://www.kaggle.com/brtknr/sushisandwich : dataset for sandwich images.
https://www.kaggle.com/meemr5/vadapav : dataset for vada pav & burger images.
(Cleaned the dataset for corrupt image files)

Total images dataset = ~3000. Split that into training/testing as 80% - 20% of the total image dataset.

Split that 80% of training data as 40% sandwich images - 40% burger images - 20% vadapav images.

Finally had 963 sandwich images, 1004 burger images and 492 vadapav images. Total size of training dataset = 2459 images. Total size of testing dataset = 608.

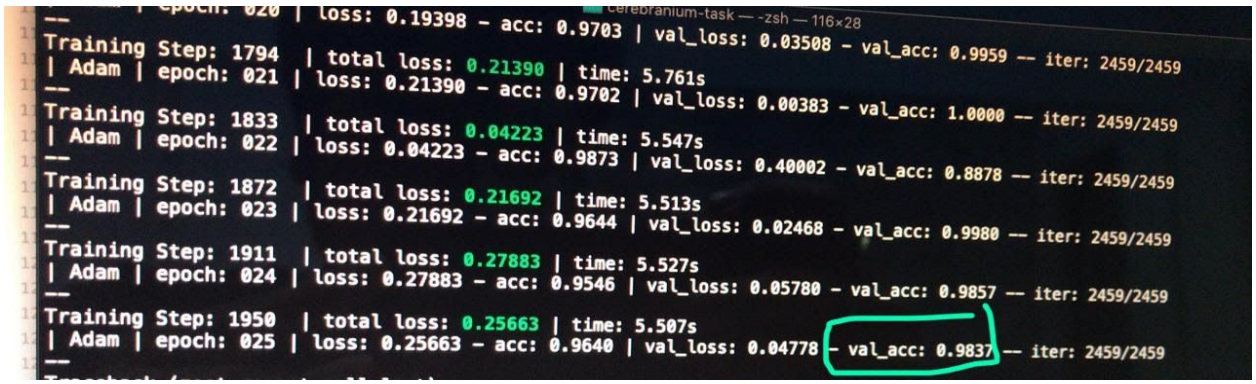


Used 20% (490 images) of training data for cross-validation.

Performance Metrics

Before starting, I always need to clearly understand our problem at hand, then choose a metric to measure the performance of our model & a loss function on which we intend to do our optimization.

The problem is of multi-class single-label image classification, for which I have chosen **accuracy** as the main metric and **categorical cross-entropy** as the loss function on which I intended to optimise.



The reason for choosing categorical cross-entropy and not sparse categorical cross-entropy was that sparse categorical cross-entropy is used when output is NOT one-hot encoded i.e. (1,2,3 etc.) and categorical cross-entropy when output is one-hot encoded i.e. ([1,0,0] or [0,1,0] or [0,0,1]).

High-level Model Architecture

filters: Mandatory Conv2D parameter is the number of filters that convolutional layers will learn from. It is an integer value and also determines the number of output filters in the convolution. Here we are learning a total of 32 filters (see marking '1' in the screenshot below) in the first layer.

max-pooling: We use max-pooling (see marking '2' in the screenshot below) i.e. 5x5 to reduce the spatial dimensions of the output volume. (As far as choosing the appropriate value for no. of filters, it is always recommended to use powers of 2 as the values.)

kernel_size: This parameter determines the dimensions of the kernel. Common dimensions include 1×1, 3×3, 5×5, and 7×7 which can be passed as (1, 1), (3, 3), (5, 5), or (7, 7) tuples of 2 integers or as a single integer, specifying the height and width of the 2D convolution window (this parameter must be an odd integer). We use 5x5 dimensions for the kernel (see marking '3' in the screenshot below).

activation: The activation parameter to the Conv2D class is simply a convenience parameter which allows you to supply a string, which specifies the name of the activation function you want to apply after performing the convolution. We have consistently used the ReLu function across the model, except in the final layer.

```
convnet = input_data(shape=[None, IMG_SIZE, IMG_SIZE, 1], name='input')

convnet = conv_2d(convnet, 32, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = conv_2d(convnet, 64, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = conv_2d(convnet, 128, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = conv_2d(convnet, 64, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = conv_2d(convnet, 32, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = fully_connected(convnet, 1024, activation='relu')
convnet = dropout(convnet, 0.8)

convnet = fully_connected(convnet, 3, activation='softmax')
convnet = regression(convnet, optimizer='adam', learning_rate=LR, loss='categorical_crossentropy', name='targets')

model = tflearn.DNN(convnet, tensorboard_dir='log')
```

Let's talk about the final layer. Now the important part is the choice of the output layer. The usual choice for multi-class single-label classification is the softmax layer. The softmax function is a generalization of the logistic function that "squashes" a K-dimensional vector-**z** of arbitrary real values to a K-dimensional vector $\sigma(\mathbf{z})$ of real values in the range [0,1] that add up to 1.

Assume our last layer (before the activation) returns the numbers z = [1.0, ,4.0, 2.0]. Every number is the value for a class. Lets see what happens if we apply the softmax activation. Using the softmax activation function at the output layer results in a neural network that models the probability of a class. A consequence of using the softmax function is that the probability for a class is not independent from the other class probabilities. This is nice as long as we only want to predict a single label per sample.