INDIVIDUAL REPORT- RITIKA AGARWAL

INTRODUCTION

We selected the Yelp dataset for our project.
Users who upload images to Yelp, are given an option of tagging the images which most users do not complete. This leaves many images without tags. Yelp wants to come up with a method that can automatically tag the uploaded images.

We decided to take up this challenge and to use convolutional neural networks to classify the images we will be experimenting with different parameters using PyTorch, TensorFlow, and Keras frameworks.

It consists of 280,000 images which have been uploaded by people and tagged by the users as 'FOOD', 'DRINK', 'MENU', 'INSIDE', 'OUTSIDE'.

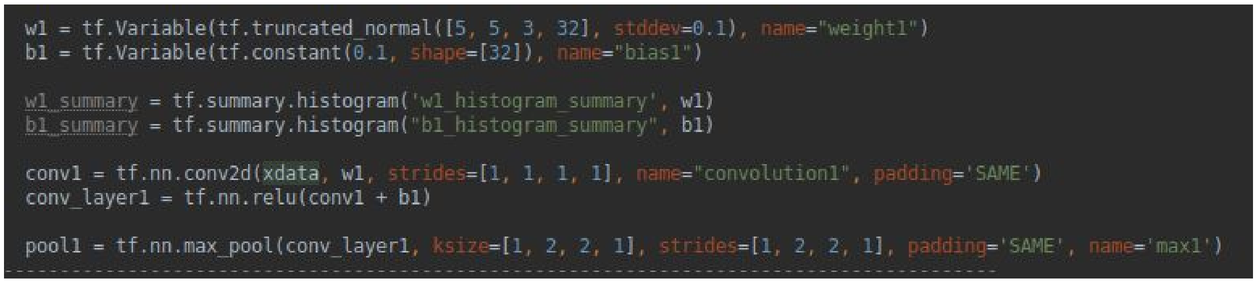
We used Convolutional Neural Networks to classify the images.
Our first task was pre-processing the data. The images are of different sizes as well as different number of channels.
We resized these images to 32*32 images with channel size 3. Some of the mages in the dataset had a channel size of 4, which skipped. The fourth channel is an alpha channel.
We skipped these images. We normalized the data, encoded it and saved the data in the form of npy files.

After the cleaning, we split the data into training and testing.
Some of the data pre processing was done together as a team.

INDIVIDUAL WORK:
Label encoding using pd.get_dummies(), Normalization, Tensorflow.

Tensorflow:
After preprocessing our data was of the shape [100000,32,32,3]
This is ideal for tensorflow as it requires data in the format of [No if images, Height, Width, No of channels]
We had 70% data in the training and 30% as testing
Due to the large size in testing data, it gave me an error , regarding the resource allocation.
Hence I decided to split the testing data into batches and calculate the accuracy depending on the mean for all the test batches.

To start of with, I used to a network with 2 convolution layers and two fully connected layers .


```
w1 = tf.Variable(tf.truncated_normal([5, 5, 3, 32], stddev=0.1), name="weight1")
b1 = tf.Variable(tf.constant(0.1, shape=[32]), name="bias1")

w1_summary = tf.summary.histogram('w1_histogram_summary', w1)
b1_summary = tf.summary.histogram("b1_histogram_summary", b1)

conv1 = tf.nn.conv2d(xdata, w1, strides=[1, 1, 1, 1], name="convolution1", padding='SAME')
conv_layer1 = tf.nn.relu(conv1 + b1)

pool1 = tf.nn.max_pool(conv_layer1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', name='max1')
```

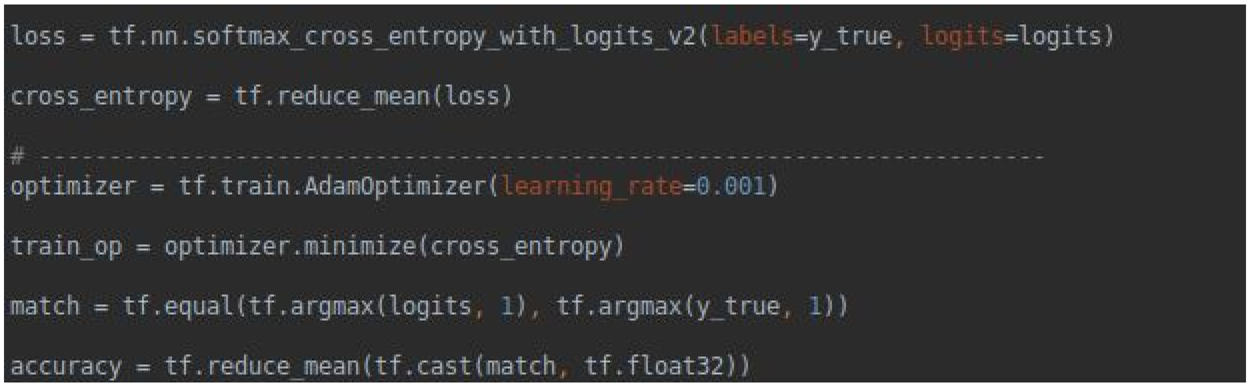
This is the first layer that I used.
I created a weight and bias variable. The weight is initialized from a truncated normal distribution with standard deviation of 0.1. The kernel size used is 5*5, 3 for the number of channels and 32 for the number of feature maps.
The bias takes a constant value of 0.1.

I then used the conv2d inbuilt tensorflow function which Computes a 2-D convolution given 4-D input and filter tensors. The 4D input is my data which is of the shape [image size, H,W, channels].
The weight initialized above is the filter tensor. Padding - SAME keeps the image size the same even after convolution by adding proportional amount of padding.

I then used Relu as the activation function on W*p + b. We require an activation function for each convolution layer to increase number linearity in the layers. ReLu is the most commonly used activation and is computationally efficient.
I used max pooling with *K size and stride as [1,2,2,1].*
These numbers denote values for every dimension. We need the kernel to move along for height and width hence we use the former mention sizes.
For the first layer I used 32 feature maps, followed by 64 feature maps and then 128 feature maps for the final convolution layer.

I then had to flatten my final convolution layer. The size of the final pooling layer for my three layered network is [batch size, 4,4,128]. Hence I flatten it to take the batch size and 4*4*128.

I have two fully connected layers with , with weights which take the size of the input which is 4*4*128 from the fully connected layer and the number neurons.



```
loss = tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_true, logits=logits)

cross_entropy = tf.reduce_mean(loss)

# -----------------------------------------------------------------------
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)

train_op = optimizer.minimize(cross_entropy)

match = tf.equal(tf.argmax(logits, 1), tf.argmax(y_true, 1))

accuracy = tf.reduce_mean(tf.cast(match, tf.float32))
```

I used cross entropy as the loss function and softmax as the activation for the last layer.

I used Adam as the optimizer and then checked the accuracy for the testing data.
Tf. argmax, gives the index of the max value and equal compares the two indexes resulting in a True or False value. I cast it to 0s and ones and then take the mean to find the accuracy.

In order to get the best accuracy I tried multiple parameters by changing batch size, adding dropout layers, number of iteration, different optimizers and leaning rates.

In order to ensure that I was not overfitting my model, I used a dropout probability of 0.3 and checked my accuracy on the training as well as the testing data. The accuracy of the training data is higher than the testing.

10 epochs - NO DROPOUT

TRAINING BATCH SIZES	TIME	ACCURACY	LOSS
50	220.79	0.83	0.5100822
100	160.54	0.79	0.604623
500	128.61	0.72	0.7603003
1000	134.83	0.81	0.74499595
5000	137.6	0.69	7.083618

As the batch size increases , the accuracy decreases and the time taken decreases, except for a batch size of 1000 which gave me a higher accuracy.
In order to to use a batch size which was efficient in terms of time and accuracy I used a batch size of 1000 for the rest of my tests except the final.

OPTIMIZER	TIME	ACCURACY	LOSS
ADAMS	134.83	0.81	0.74499595
RMSPROP	138.72	0.79	0.63258666
GRADIENT DESCENT OPTIMIZER	139.32	0.66	0.9261319

I tried 3 different optimizers, Adam Optimizer gave me the best accuracy.

ADAM OPTIMIZER

LR	TIME	ACCURACY	LOSS
0.001	134.83	0.81	0.74499595
0.05	134.8	0.67	0.92646533
0.01	131.66	0.7200001	0.75439185

A learning rate of 0.001 gives me the highest accuracy.

10 ITERATIONS - ADAM - 1000 BATCH

DROPOUT PROB	TIME	ACCURACY	LOSS
0.5	134.05	0.66	0.90103066
0.2	138.85	0.73	0.82522136

I then added dropout probabilities, but this decreased the accuracy.
I wanted to experiment a little further with our model so I increased the number of neurons in our final layers.

NEURONS	TIME	ACCURACY	LOSS
2000	148.3	0.84	2.9962485

With 2000 neurons, the accuracy of our model increased. There was a possibility of overfitting, hence I tested the training as well as the testing set and I saw that the accuracy remained the same. Hence we were not overfitting the data.

I then added one more convolution layer to see if our accuracy would increase.

LAYERS	TIME	ACCURACY	LOSS
3	163.11	0.84999	6.324423

There was a very slight increase in accuracy.

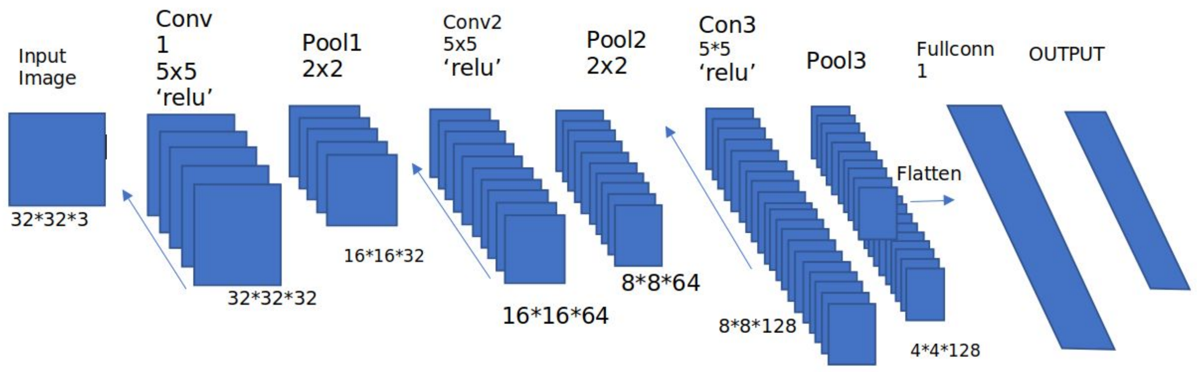
In order to finalize our model, we changed the batch size to 100 and ran it for 100 iterations

LAYERS	TIME	ACCURACY	LOSS
3 layers	2003.0	0.91	0.04989095

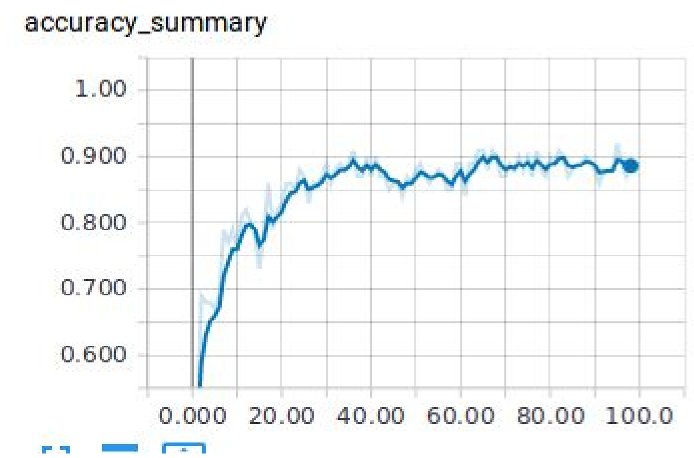
.
Our accuracy considerably increased which is the best accuracy that we got.

In order to regularize our data , I then used the same parameters but also added in a drop out probability of 0.3. This gave us an accuracy of 0.86 and loss 0.103.

FINAL NETWORK

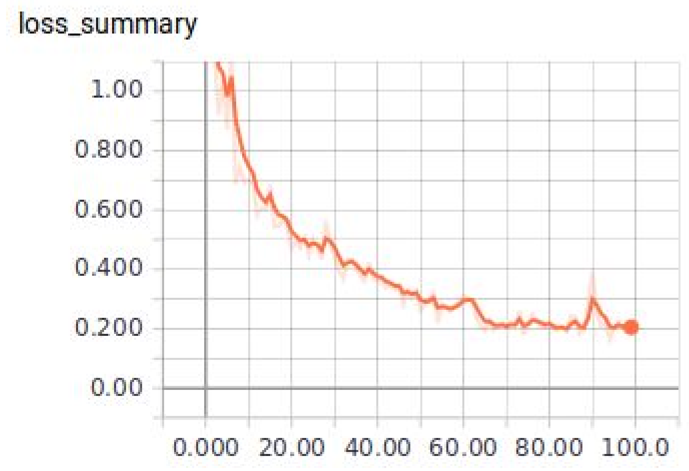


TEST ACCURACY



This graph shows the testing accuracy over 100 iterations for our final model. As you can see, there is slight fluctuation but overall there is an increase in the accuracy. The fluctuation could be reduced by normalizing the data furth and some data augmentation.

TRAIN LOSS



This show the training loss over 100 iterations for our final model. Again, there is slight fluctuation which especially around the 90th iteration which can be reduced by further data processing and normalizing.

CONCLUSIONS:

The lower the learning rate, higher was my accuracy.I got the best model when I ran 100 iterations using batch size of 100 and 3 layers.The best accuracy I got is 91% but however after I added the drop out nodes, to prevent overfitting, I got an accuracy of 86%.
 Adam Optimizer worked the best for me.
Tensorflow is easy to use but however hard to debug, tensorboard is a convenient tool for visualizations.
For the future, it would make more sense to ensure that the data is is more evenly distributed by taking samples of the data from all classes.

PERCENTAGE OF CODE FROM THE INTERNET:
70% of the code is from the internet.

REFERENCES:
UDEMY course - complete-guide-to-tensorflow-for-deep-learning-with-python
https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c
Tensorflow Documentation