Machine Learning 2 - Final Project

Group 4
Ritika Agarwal, Tanaya Pole, Selvyn Perez

Introduction

For our project, we are solving the Yelp Dataset Challenge. Yelp provided us with data without instructions. The goal of their challenge is to use their data in innovative ways that might be able to help them with their research. Their data consists of images and a json file with information on the images such as photo id and labels.

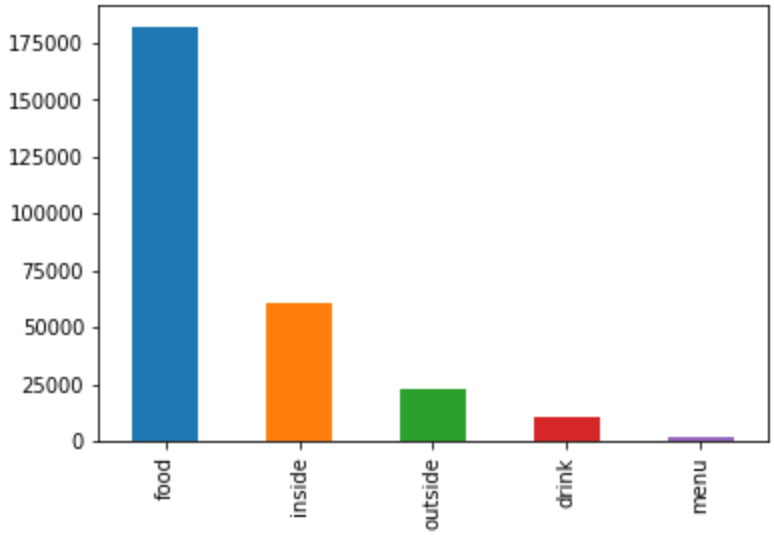
Yelp is a search engine service powered by a crowd sourced review forum. It has reviews and recommendations about nightlife, shopping, restaurants. Users who upload images to Yelp, are given an option of tagging the images which most users do not complete. This leaves many images without tags. Yelp wants 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.

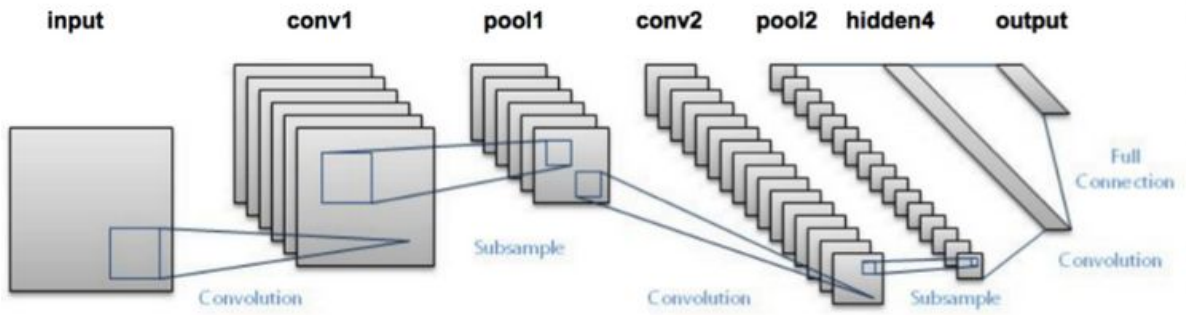
Description of the data set

We were provided with a set of photos (200,000+), a json file with photo id and their labels (food, drink, inside, outside, menu), and a csv file with photo id and 99 columns of data for each picture. We do not have any information on what the data in each of the columns represents. Since we do not know what they are, we decided not to use them. Instead, we created a new csv file with the photo id and merged it with the labels from the json file. We will be using the new processed csv file (with photo ids and their respective labels) and the images.

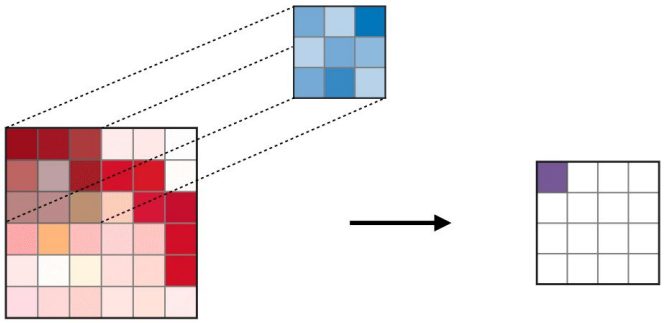
Below is a plot of the labels in our data set with their respective frequency. ‘Food’ has the highest number of photos in comparison to the others while ‘menu’ has the least number of photos.



Deep Learning Network and Training Algorithm



We are using convolutional neural networks because they are a type of neural network that are commonly used for image recognition and classification. The layers generally consist of an input image, convolutions, pooling, and a fully connected matrix, as you can see in the image above. The convolution layer scans the input image using filters (in blue in the figure below) with a stride and with respect to its dimensions. The output is called a feature map.



(https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks)

The pooling layer basically down samples the feature map and can be done by selecting the maximum or average from the view. Given a stride, the maximum is selected keeping in mind the dimensions of the max pooling kernel. The output of the pooling layer is then flattened in the fully connected layer, where the input is connected to the number of neurons.

The frameworks we have chosen to work with are PyTorch, TensorFlow, and Keras. The parameters we will be looking at are computational time, network layers, learning rate, training data loss, testing data accuracy, overfitting, batch size, number of epochs, and optimizer.

Experimental setup

Data Preprocessing:

The images that we downloaded from the Yelp challenge all have different dimensions. In order to use a convolutional neural network, we decided that we needed to resize the images to 32x32. This process was very time consuming as we had to leave the program running overnight to get it all resized.

The yelp dataset came with roughly 280,000 images. The folder size storing all of these images was over 3GBs. We decided to upload these images to the google cloud storage bucket. Google has a utility that can be installed locally and allows you to upload/update files to your bucket. We used this utility to upload all of the images. This also took a long time, and we had to use the terminal to upload it into the bucket overnight. After they were uploaded, we needed to set the permissions to allow public access to view them. This is an important step because it allowed us to write our models without needing to have the images on our directory. All the images would be retrieved from the cloud.

The first pre-processing step we had to complete was to map the json file with the csv image file. This was because the json file had the image labels which were what we were going to try to predict while the csv had the image ids which would be needed to retrieve them from the bucket.

Next, using urllib we were able to retrieve each individual image using the picture_id and building a response from the url. We saved the images and labels into separate .npy files and also uploaded them to the bucket and made them public. From here on, we were able to directly reference the .npy files in each of our python files in order to speed up the run time significantly.

We then normalized the images, by subtracting each each pixel value by the minimum pixel value, and dividing by max - min value.

min_val = np.min(x)
max_val = np.max(x)
x = (x-min_val) / (max_val-min_val)

Where, x: input image data in numpy array [32, 32, 3]

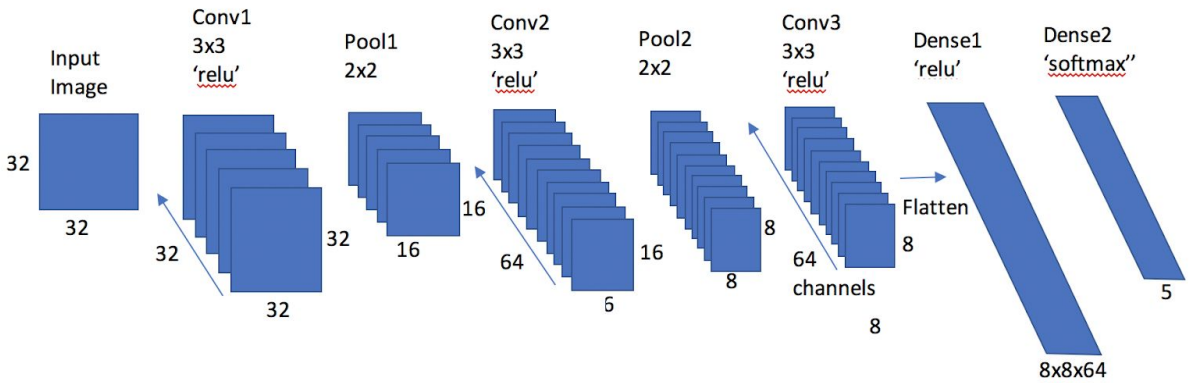
We use pd.get_dummies to encode our data and and then saved the data as .npy files which we can just import while running our models. Once our preprocessing was complete, we split the data into train (70%) and test (30%) using Scikit learn.

Up until here the pre-processing for every framework was the same , we can directly load the cleaned data and run our models on it. The cleaned data has the form of [70000,32,32,3] and the labels are of form [30000,5]. We have 5 labels, which include [food, drink, menu, inside, outside] .

Results

Keras:

Keras is an open source neural network library that runs on top of Tensorflow and Theano. It is user friendly, allows for fast results, and is commonly used for convolutional neural networks. The following shows the architecture of the convolutional neural network we used in Keras.



In Keras, it is not needed to split the data beforehand, but because we had already split our data using Scikit Learn, we decided to input our already normalized and split data into the model.

In our convolutional layers, we used ReLu as our activation function because it is one of the most common activation functions, and it also speeds up training. Any negative numbers are set to 0. The dense layers are the fully connected matrices. In the first dense layer, we used ReLu again because it is computationally efficient. We used softmax in the second because softmax outputs a probability range for each label, which is how we want our output to look like. We need to be able to see the probability of each label for each image. We used cross entropy to calculate the loss because because we are using a classification model.

We decided to use 3 Convolutional layers with two poolings because it gave us the highest accuracy. The following is a table that shows the other architectures we considered. For each one, the batch size is 100, epochs is 5, optimizer is rmsprop.

Layers	Accuracy	Loss
1 Conv, 1 Max Pool	0.8663	0.4296
2 Conv, 2 Max Pool	0.9033	0.2847
3 Conv, 2 Max Pooling	0.9045	0.2828
3 Conv, 3 Max Pooling	0.8948	0.3000

Here, you can see that the highest accuracy with the lowest loss is for 3 convolutional layers with 2 max poolings. Now that we have decided on the architecture, we decided to experiment with optimizers. The optimizers we chose to investigate are Rmsprop, Adam, SGD, and Adagrad.

Optimizer	Accuracy	Loss	Time (s)
rmsprop	0.8923	0.3177	71.3795
Adam	0.8929	0.3057	75.2057
SGD	0.7605	0.6912	71.3718
Adagrad	0.8360	0.4748	69.3098

Adam and rmsprop have very similar accuracies and losses. Because Adam took the longest time to run, we decided to go with rmsprop.

We also experimented with several learning rates to decide on an optimal one.

Learning rate	Accuracy	Loss	Time(s)
0.001	0.8923	0.3177	71.3794
0.0001	0.8542	0.4189	71.1081
0.01	0.6548	5.5634	72.3446
0.005	0.8073	0.5508	74.1131

All of the learning rates took similar amounts of computational time. The accuracy for 0.001 was the highest so we decided to choose that one.

Batch Size	Accuracy	Loss	Time(s)
100	0.8923	0.3177	71.3794
500	0.8541	0.4046	61.5452
1000	0.8247	0.4973	64.0554
50	0.9040	0.2931	97.5911

The accuracy for the batch size of 50 was the highest, however it was significantly slower to run. The accuracy of a batch size of 100 was slightly lower, but it ran about 30 seconds faster, so we decided to keep that as the batch size.

We decided to select 5 epochs because it took the least amount of time and the accuracies for for a higher number of epochs did not change significantly.

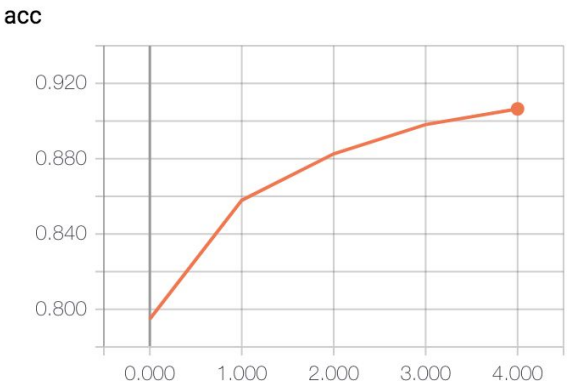
Epochs	Accuracy	Loss	Time(s)
5	0.8923	0.3177	71.3794
100	0.8904	1.1933	752.6329
500	0.8925	1.6946	3521.7870

Last, to check for overfitting, we decided to add a dropout layer.

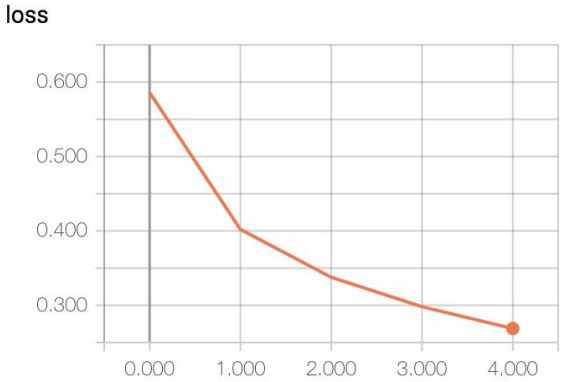
Dropout	Accuracy	Loss	Time(s)
0.0	0.8923	0.3177	71.3794
0.2	0.8839	0.3387	74.8656
0.4	0.8937	0.31295	72.7018

Adding a dropout layer and increasing the percentage of dropout didn't affect the accuracy much. This showed that our model was not overfitting.

Using tensorboard, we graphed our accuracy and losses.



Here, you can see that as the number of epochs increases, the accuracy also increases.



As the number of epochs increases, the training loss decreases. This is because loss is minimized during training.

Overall for CNN using keras, the test accuracy was 0.8923 with a loss of 0.3177. This network took 71.4 seconds.

Tensorflow:

After pre -processing our data, the shape of our data for the was [70000,32,32,3] for training and
[30000, 5] for the labels.

This is ideal for tensorflow since it requires data to be in the format of [Number of images, Height , Width, No of channels]
To start of with, we used 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 we used. We 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.

We then used the conv2d inbuilt tensorflow function which computes a 2-D convolution given a 4-D input and filter tensors. The 4-D 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.

We 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. We 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 we used 32 feature maps, followed by 64 feature maps and then 128 feature maps for the final convolution layer.

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

We 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))
```

We used cross entropy as the loss function and softmax as the activation for the last layer. Softmax gives the probability that the image belongs to a particular label. We 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. We cast it to 0s and ones and then take the mean to find the accuracy.

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

10 epochs

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 we used a batch size of 1000 for the rest of our tests.

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 us the best accuracy.

Using Adam Optimizer we tried multiple learning rates as follows

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 us the highest accuracy.

We wanted to experiment a little further with our model so we 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.

We then added one more convolution layer to see if our accuracy would increase.
3 LAYERS

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.

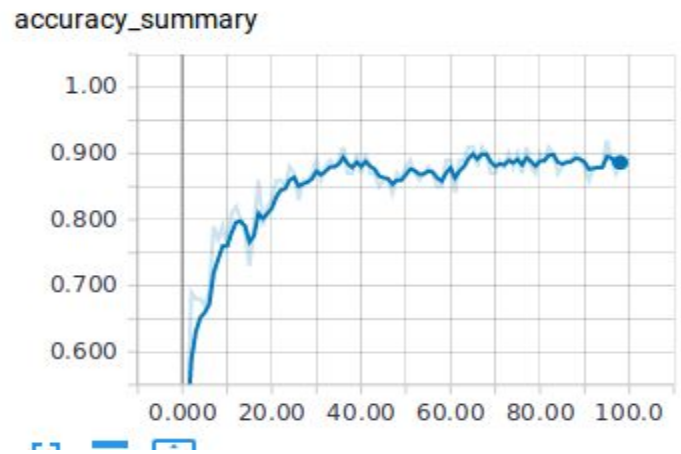
LAYERS	TIME	ACCURACY	LOSS
3 layers	2003.0	0.91	0.04989095

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

In order to ensure that we were not overfitting our model, we used a dropout probability of 0.3 and checked the accuracy on the training as well as the testing data. The accuracy of the training data is very slightly higher than the testing.
This gave us an accuracy of 0.86.

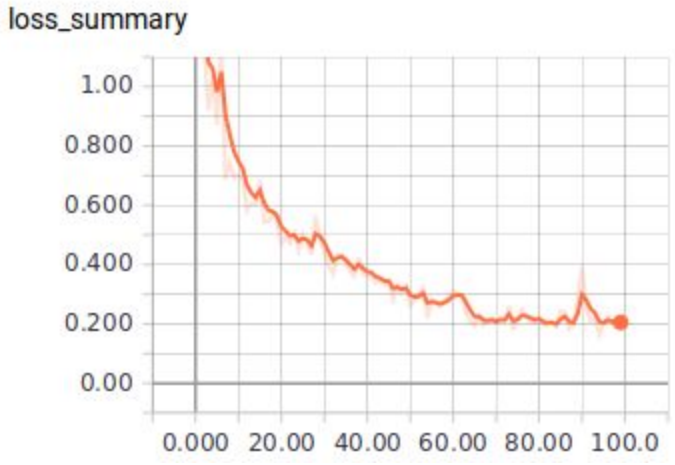
```
test_accuracy 0.86
Loss :0.103167966
Epoch number: 99 Epoch time: 2365.35
```

TESTING ACCURACY:



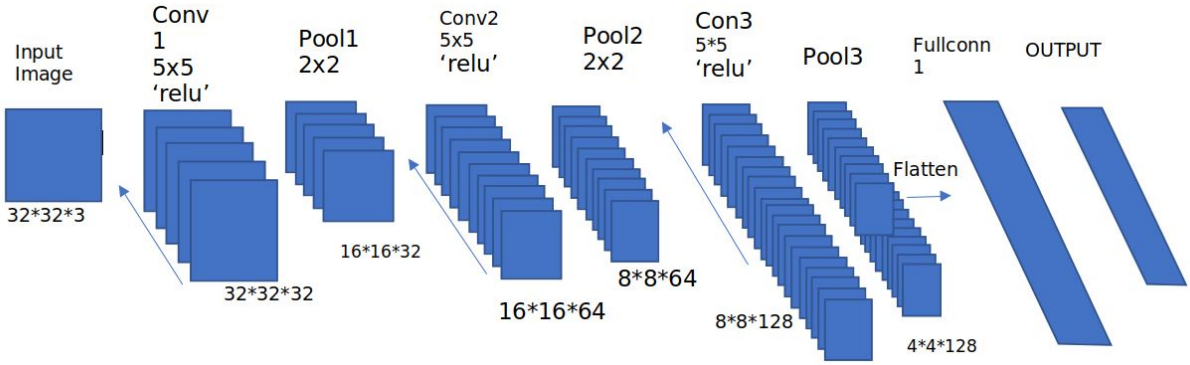
This graph shows the testing accuracy over 100 iterations for our final model. As you can see, there is slight fluctuation but overall there is an increase in the accuracy. The fluctuation could be reduced by normalizing the data further and some data augmentation.

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.

FINAL NETWORK



OPTIMIZER: Adams
LOSS FUNCTION: Cross entropy
LEARNING RATE:0.001
EPOCHS:100
BATCH SIZE: 100
DROPOUT PROBABILITY:0.3
ACCURACY:0.86

We were facing memory errors saying Resource Exhausted , hence we carried out the testing in batches of 100 as well and calculated the mean. Due to this time required for every epoch was higher.

Pytorch:

PyTorch is an open-source machine learning library for Python, based on Torch, used for applications such as natural language processing. Pytorch is a good framework for running a CNN but the performance is slower. After getting the model to read in the npy data, we needed to split the data in 70/30 train/test. Then the model was ready to train. For computing the loss, we used the CrossEntropyLoss() function and used the SGD optimizer function. Also, for the purposes of helping the performance, we ran the model using the GPU. We wanted to how our accuracy would change if we removed the Convolution layers from these runs.
We ran three different models switching the layers, batch size and number of epochs. For the first run, we used the following parameters with one layer using the Softmax activation function:

```
input_size = 3072
hidden_size = 500
num_classes = 5
num_epochs = 100
batch_size = 100
learning_rate = .0001
```

Results for first run:
Epoch [100/100], Step [700/700], Loss: 0.7141
Accuracy of the network on the 100000 test images: 76 %
Accuracy of drink : 42 %
Accuracy of food : 95 %
Accuracy of inside : 63 %
Accuracy of outside : 32 %
Accuracy of menu : 15 %

To start, we felt we could run using a learning rate of .0001 and start with a larger batch size. As the results show, the we had a 76% accuracy using this method and a final loss of .7141.

For the second run, we used the following parameters:

```
input_size = 3072
hidden_size = 500
num_classes = 5
num_epochs = 100
batch_size = 10
learning_rate = .0001
```

Results for second run:
Epoch [100/100], Step [7000/7000], Loss: 0.5483
Accuracy of the network on the 100000 test images: 81 %
Actual: food food inside food inside food food food food food
Predicted: food inside inside food inside food food food food food
Accuracy of drink : 48 %

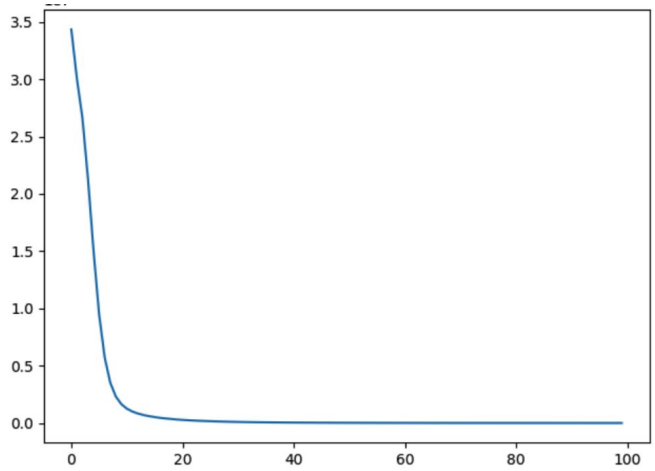
Accuracy of food : 91 %
Accuracy of inside : 77 %
Accuracy of outside : 32 %
Accuracy of menu : 50 %

By reducing the batch size significantly, the accuracy of the model increased significantly. The final loss was also reduced significantly, down to .5483.

For the final run, we added a couple more layers to the network architecture, still using the softmax function as the activation:

```
input_size = 3072
hidden_size_1 = 500
hidden_size_2 = 100
num_classes = 5
num_epochs = 100
batch_size = 20
learning_rate = .0001
```

Epoch [100/100], Step [3500/3500], Loss: 1.1605
Accuracy of the network on the 100000 test images: 65 %
Actual: food inside food food food food inside inside inside food food food food food food menu food food inside food
Predicted: food
Accuracy of drink : 39 %
Accuracy of food : 100 %
Accuracy of inside : 80 %
Accuracy of outside : 46 %
Accuracy of menu : 23 %
RUNNING TIME: 529.432516098
What we found was that the best model was when running on one layer and using the a small batch size. The performance suffered significantly when adding more layers using pytorch. As our later results would see for the other frameworks, this would not be the case when adding more layers to them.



Summary and conclusions

Our group decided to use 3 different frameworks for this project: Pytorch, Keras, and Tensorflow. We expected that all 3 frameworks would produce similar results since we were planning on using convolutional neural networks. So for 1 framework, Pytorch, we ran our model without a Convolution Layer and strictly used a Linear layer. This resulted in all 3 runs being lower in accuracy than when using Keras or Tensorflow for which we used convolutional layers for. The strongest accuracy was 81% with Pytorch for which our loss was .54. This was decent performance but as noted, the other frameworks using convolutional layers performed better.

Our next model, we used Tensorflow. The best accuracy we had was 91% which was a much better accuracy than Pytorch, but it took about 20 minutes to run. We then used 3 convolutional layers along with a dropout rate of .3 and a batch size of 100. This decreased our accuracy to 86%.Using the Adams optimizer helped produce the best accuracy along with keeping the batch size from 100 to 1000.

For the Keras framework, we got similar results as the Tensorflow model. We produced the best accuracy when using 3 convolutional layers and 2 max pooling layers without a dropout layer. We also received the best accuracy using the rmsprop optimizer. The best accuracy for all the runs using the Keras model was 89%. In conclusion, we learned that using these frameworks have different benefits, for example with Tensorflow and Keras we can use Tensorboard for visualization and Keras has a simpler implementation method of the convolutional neural networks than both Tensorflow and Pytorch do. Without using convolutional layers, as we did for Pytorch, we learned that our accuracy was not as high than when using different methods like convolutions, pooling, and dropouts for the other frameworks.

FUTURE WORK:

As our data was not evenly distributed, we can improve our model, by data sampling. The number of images belonging to each label should be proportionate. We can also use other measures apart from accuracy and loss to test our model, such as precision and recall.

References

- https://pytorch.org/docs/0.3.1/nn.html#linear
- https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/
- https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- https://blog.algorithmia.com/convolutional-neural-nets-in-pytorch/
- UDEMY course - complete-guide-to-tensorflow-for-deep-learning-with-python
- https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c
- Tensorflow Documentation
- https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/4_Utils/tensorboard_basic.py
- Deep Learning with Python by Francois Chollet
- https://keras.io/callbacks/