Deep Neural Networks: GPU Task 1

What this task covered:

- Train neural network & Image classification
  Teach a neural network to separate images into groups/classes, to recognize Louie or Not Louie.

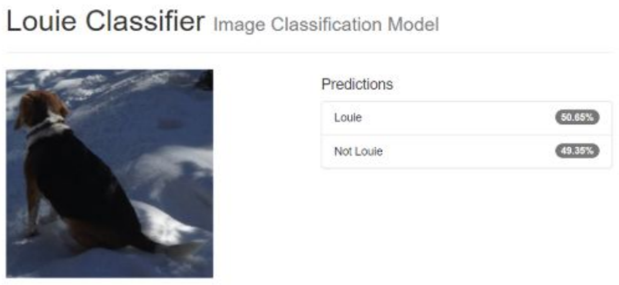
What are the steps:

1. Create a model by choosing a small dogs dataset
2. Set num_epochs: 2 - 100
3. Select network configuration: AlexNet
4. Train the model
5. Test a random image and check the performance

What I learned:

- The more epoch, the better performance, since the Louie classifier will be more accurate
- The training process include:
    - Forward propagation yields an inferred label for each training image
    - Loss Function used to calculate difference between know label and predicted label for each image
    - Weights are adjusted during backward propagation
    - Repeat the process

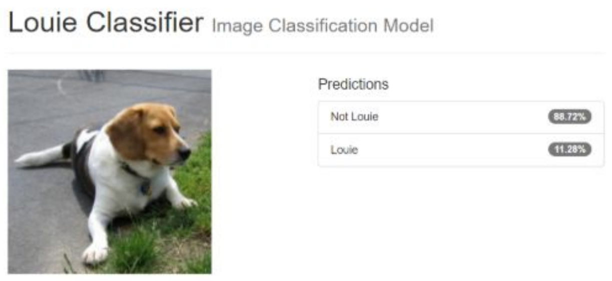
Model Performance:



Louie Classifier Image Classification Model

Predictions

Louie	50.65%
Not Louie	49.35%



Louie Classifier Image Classification Model

Predictions

Not Louie	88.72%
Louie	11.28%

Big Data: GPU Task 2

What this task covered:

- Combined Deep Neural Networks and Big Data to train a image classification network Apply a large dataset to previous classification and create a network on new data to correctly classify images of dogs and cats

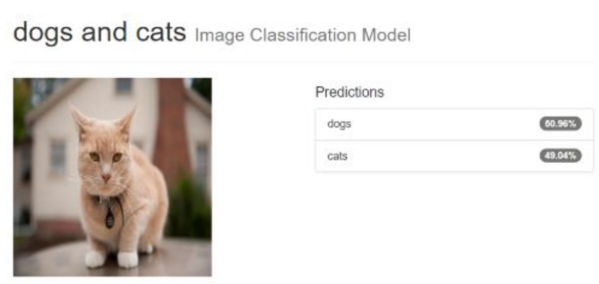
What are the steps:

1. Loading and organizing the dataset
    - Standardize them to the same size to match what the network you are training expects. We'll be training AlexNet again which was designed to take an input of 256X256 color images
    - Split them into two datasets, where 75% of each class is used for training and 25% is set aside for validation
2. Set num_epochs: 5
3. Select network configuration: AlexNet
4. Train the model
5. Test a random image and check the performance

What I learned:

- The training dataset will be used in the way we saw when we trained our Louie classifier; forward propagate an image through the network, generate an output, assess the loss, backward propagate the loss back through the network to update weights
- The validation dataset will be used to assess performance on new data. Validation data is fed through the network to generate an output, but the network does not learn anything from the data
- Inference is the process of making decisions based on what was learned. The power of our trained model is that it can now classify unlabeled images.

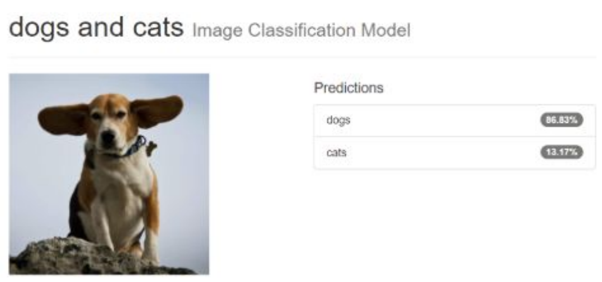
Model Performance:



dogs and cats Image Classification Model

Predictions

dogs	50.96%
cats	49.04%



dogs and cats Image Classification Model

Predictions

dogs	86.83%
cats	13.17%

Deploying our Model: GPU Task 3

What this task covered:

- Deploy trained networks into applications to solve problems in the real world
  Create a doggy door that's exclusive to dogs, that deploy previous trained model into a simulator

What are the steps:

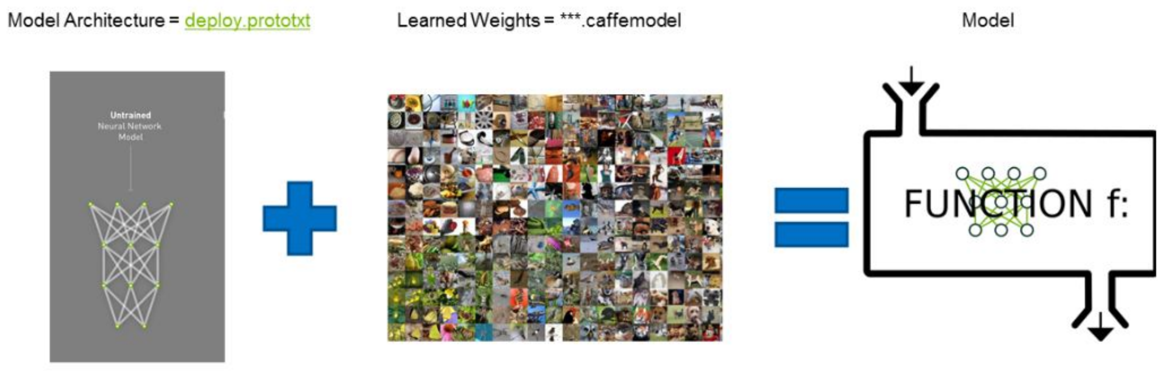
1. Get the File path to Architecture and Weights
    - The architecture is the file called deploy.prototxt
    - The weights are in the most recent snapshot file snapshot_iter_#.caffemodel
2. Create a "Classifier" object called "net"

```
# Initialize the Caffe model using the model trained in DIGITS
net = caffe.Classifier(ARCHITECTURE, WEIGHTS,
                       channel_swap =(2, 1, 0), #Color images have three channels, Red, Green, and Blue.
                       raw_scale=255) #Each pixel value is a number between 0 and 255
                       #Each "channel" of our images are 256 x 256
```

3. Creating an Expected Input: Preprocessing
4. Forward Propagation
   prediction = net.predict([grid_square])
5. Generating a useful output: Post processing

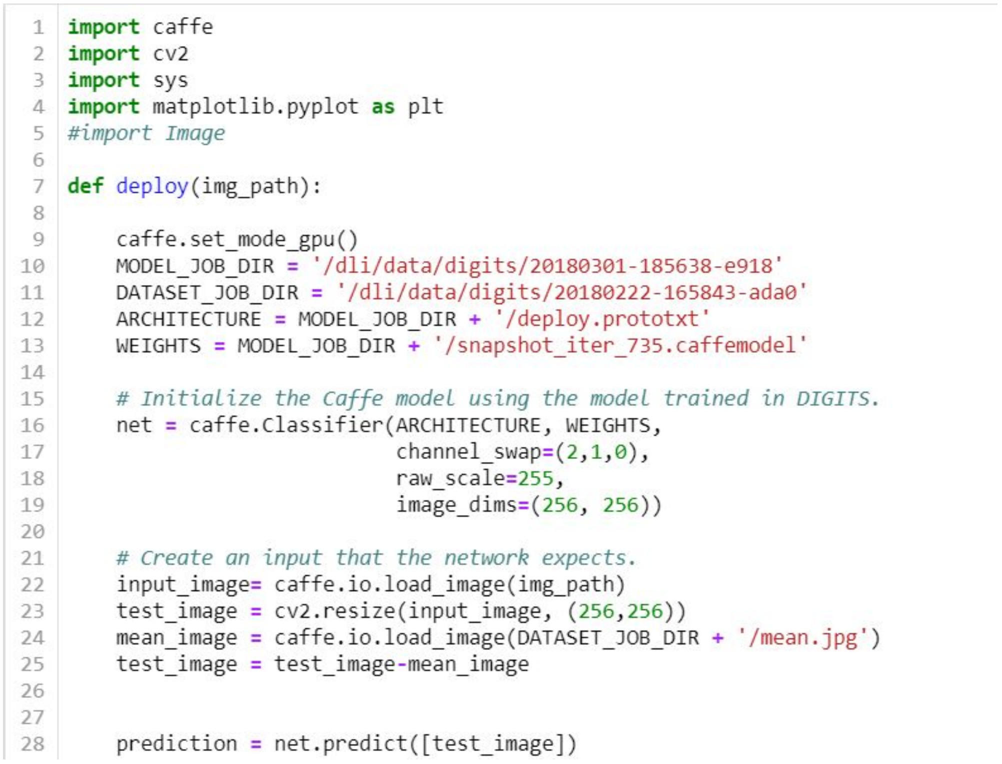
What I learned:

- The model, which the function can be used within an application contains the model architecture and learned weights

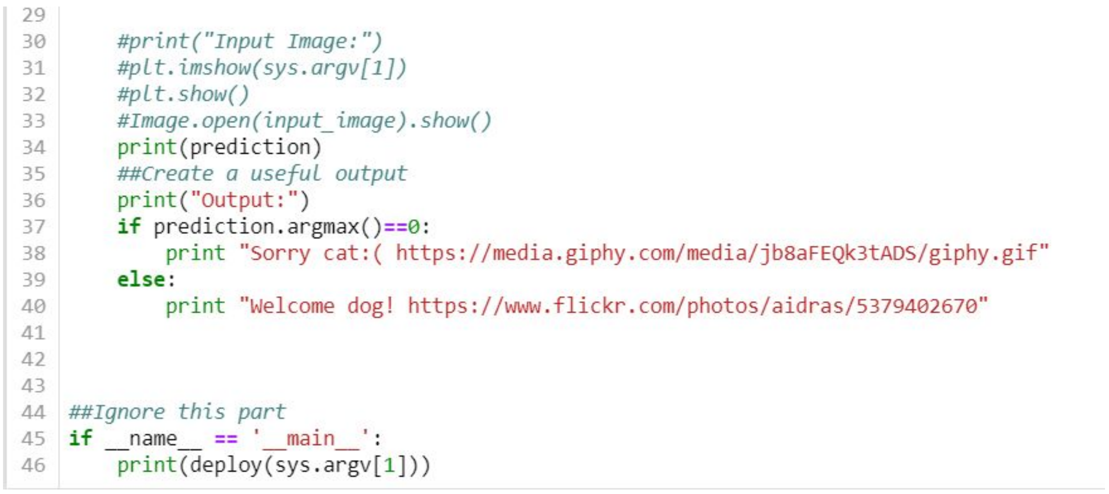

Model Architecture = deploy.prototxt
Learned Weights = ***.caffemodel
Model
Untrained Neural Network Model
FUNCTION f:

- Whatever was done prior to training must be done prior to inference

Generated Codes:



```python
import caffe
import cv2
import sys
import matplotlib.pyplot as plt
#import Image

def deploy(img_path):

    caffe.set_mode_gpu()
    MODEL_JOB_DIR = '/dli/data/digits/20180301-185638-e918'
    DATASET_JOB_DIR = '/dli/data/digits/20180222-165843-ada0'
    ARCHITECTURE = MODEL_JOB_DIR + '/deploy.prototxt'
    WEIGHTS = MODEL_JOB_DIR + '/snapshot_iter_735.caffemodel'

    # Initialize the Caffe model using the model trained in DIGITS.
    net = caffe.Classifier(ARCHITECTURE, WEIGHTS,
                           channel_swap=(2,1,0),
                           raw_scale=255,
                           image_dims=(256, 256))

    # Create an input that the network expects.
    input_image= caffe.io.load_image(img_path)
    test_image = cv2.resize(input_image, (256,256))
    mean_image = caffe.io.load_image(DATASET_JOB_DIR + '/mean.jpg')
    test_image = test_image-mean_image


    prediction = net.predict([test_image])
```



```python

    #print("Input Image:")
    #plt.imshow(sys.argv[1])
    #plt.show()
    #Image.open(input_image).show()
    print(prediction)
    ##Create a useful output
    print("Output:")
    if prediction.argmax()==0:
        print "Sorry cat:( https://media.giphy.com/media/jb8aFEQk3tADS/giphy.gif"
    else:
        print "Welcome dog! https://www.flickr.com/photos/aidras/5379402670"



##Ignore this part
if __name__ == '__main__':
    print(deploy(sys.argv[1]))
```

Performance during Training: GPU Task 4

What this task covered:

- To run more training epochs on an existing model, analogous to a human learner studying more
- To search the hyperparameter space, analogous to a human learner responding differently to a different teaching style
- To use the results of others' research, compute, network design, and data, analogous to a human learner copying off an expert

What are the steps:

1. First intervention: "Will the model keep improving if we let it train longer?"
2. Study more: entail running more epochs from where our model last left off
3. Create a new model from pretrained model
4. Deploying expert pretrained models
   All we need to deploy one of these models are the model's architecture and weights
5. Download "pretrained model alexnet imagenet caffe" using a tool called wget
6. Work to make the output useful to a user

What I learned:

- Make Pretrained Model will :
    - Save the network architecture
    - Save what the model has "learned" in the form of the parameters that have been adjusted in each epochs
- Learning rate is the rate at which each "weight" changes during training. Each weight is moving in the direction that reduces loss at a value multiplied by the learning rate
- Four categories of levers that can be manipulated to improve performance:
    - Data - A large and diverse enough dataset to represent the environment where the model should work.
    - Hyperparameters - Making changes to options like learning rate are like changing the training "style."
    - Training time - More epochs improve performance to a point.
    - Network architecture
- When deploying award winning neural networks, not only can we use their network architecture, we can even use their trained weights, acquired through the manipulation of the four levers above: data, hyperparameters, training time, and network architecture. Without any training or data collection
- Wget is a great way of downloading data from the web directly to the server you're working on without pulling it to your local machine first.

Generated Codes:

```
!wget http://dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel
!wget https://raw.githubusercontent.com/BVLC/caffe/master/models/bvlc_alexnet/deploy.prototxt
```

```
!wget https://github.com/BVLC/caffe/blob/master/python/caffe/imagenet/ilsvrc_2012_mean.npy?raw=true
!mv ilsvrc_2012_mean.npy?raw=true ilsvrc_2012_mean.npy
```

```
#Load the image
image= caffe.io.load_image(TEST_IMAGE)
plt.imshow(image)
plt.show()

#Load the mean image
mean_image = np.load(MEAN_IMAGE)
mu = mean_image.mean(1).mean(1)  # average over pixels to obtain the mean (BGR) pixel values

# create transformer for the input called 'data'
transformer = caffe.io.Transformer({'data': net.blobs['data'].data.shape})
transformer.set_transpose('data', (2,0,1))  # move image channels to outermost dimension
transformer.set_mean('data', mu)            # subtract the dataset-mean value in each channel
transformer.set_raw_scale('data', 255)      # rescale from [0, 1] to [0, 255]
transformer.set_channel_swap('data', (2,1,0))  # swap channels from RGB to BGR
# set the size of the input (we can skip this if we're happy with the default; we can also change it later
net.blobs['data'].reshape(1,        # batch size
                          3,         # 3-channel (BGR) images
                          227, 227)  # image size is 227x227

transformed_image = transformer.preprocess('data', image)
```

Object Detection: GPU Task 5

What this task covered:

- Detect and localize objects within images
  - Combining deep learning with traditional computer vision
  - Modifying the internals of neural networks
  - Choosing the right network
- How to change a network to change its behavior and performance
- Object Detection
  - Using Deployment
  - Using Forward Propagation
  - Rebuilding from an existing neural network
  - DetectNet

What are the steps:

Part 1:

1. Bring in the model and dataset job directories to make use of the model architecture, trained weights, and preprocessing information such as the mean image
2. Using learned function: Forward Propagation
   - First, see the prediction from the top left 256X256 grid_square of our random image
   - Second, built ANYTHING around this function, that iterate over the image and classify each grid_square to create a heatmap
3. Keeping the grid square size as 256x256, but to increase the overlap between grid squares and obtain a finer classification map
4. To batch together multiple grid squares to pass in to the network for prediction

Part 2:

1. In current AlexNet Caffe model, visualize the AlexNet network
2. Replace some layers of the AlexNet network to create new functional
   - Removed a "fully connected" layer, which is a traditional matrix multiplication
   - Added a "convolutional" layer, which is a "filter" function that moves over an input matrix
3. Converting AlexNet to a Fully Convolutional Network

Part 3:

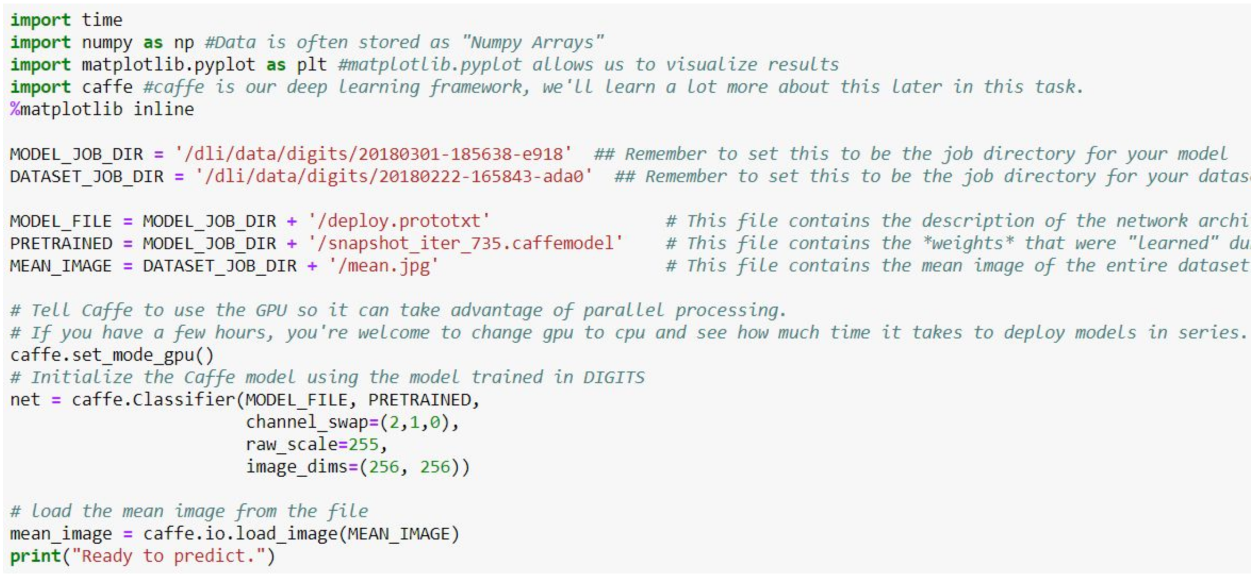
1. The challenge faced by the designers of the network we'll use was what is the most efficient and accurate way to map the input output pairings from out data

What I learned:

- Advantage of the Fully Convolutional Network:
  - Able to locate the dog with greater precision than the sliding window approach

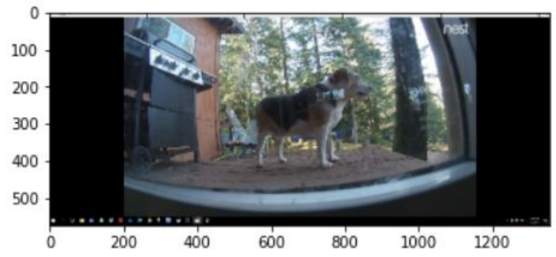
  - The total inference time has been reduced a lot
- Disadvantage of the Fully Convolutional Network:
  - Will miss parts of the dog or falsely detect pieces that are not a dog
  - Solution: Using an end-to-end object detection network designed to detect and localize dog
- Function “net.predict” passes an input, grid_square, and returns an output, prediction. Unlike other functions, this function isn't following a list of steps, instead, it's performing layer after layer of matrix math to transform an image into a vector of probabilities
- The type of layer is "softmax," where the higher the value of each cell, the higher the likelihood of the image belonging to that class
- Any deep learning solution is still just a mapping from input to output
- The advantage of this sliding window approach is that we can train a detector using only patch-based training data, which is more widely available
- One advantage of convolutional layers is that they do not only work with specific matrix size compared to the fully connected layer
- By converting AlexNet to a "Fully Convolutional Network," we'll be able to feed images of various sizes to our network without first splitting them into grid squares
- Three ingredients that make deep learning possible:
  - Deep Neural Networks
  - GPU
  - Big Data
- DetectNet is actually an Fully Convolutional Network (FCN), as we built above, but configured to produce precisely this data representation as its output.
  The bulk of the layers in DetectNet are identical to the well-known GoogLeNet network.

Generated Codes:



```
import time
import numpy as np #Data is often stored as "Numpy Arrays"
import matplotlib.pyplot as plt #matplotlib.pyplot allows us to visualize results
import caffe #caffe is our deep learning framework, we'll learn a lot more about this later in this task.
%matplotlib inline

MODEL_JOB_DIR = '/dli/data/digits/20180301-185638-e918'  ## Remember to set this to be the job directory for your model
DATASET_JOB_DIR = '/dli/data/digits/20180222-165843-ada0'  ## Remember to set this to be the job directory for your datas

MODEL_FILE = MODEL_JOB_DIR + '/deploy.prototxt'                 # This file contains the description of the network archi
PRETRAINED = MODEL_JOB_DIR + '/snapshot_iter_735.caffemodel'    # This file contains the *weights* that were "learned" du
MEAN_IMAGE = DATASET_JOB_DIR + '/mean.jpg'                      # This file contains the mean image of the entire dataset

# Tell Caffe to use the GPU so it can take advantage of parallel processing.
# If you have a few hours, you're welcome to change gpu to cpu and see how much time it takes to deploy models in series.
caffe.set_mode_gpu()
# Initialize the Caffe model using the model trained in DIGITS
net = caffe.Classifier(MODEL_FILE, PRETRAINED,
                       channel_swap=(2,1,0),
                       raw_scale=255,
                       image_dims=(256, 256))

# load the mean image from the file
mean_image = caffe.io.load_image(MEAN_IMAGE)
print("Ready to predict.")
```

```python
# Choose a random image to test against
#RANDOM_IMAGE = str(np.random.randint(10))
IMAGE_FILE = '/dli/tasks/task5/task/images/LouieReady.png'
input_image= caffe.io.load_image(IMAGE_FILE)
plt.imshow(input_image)
plt.show()
```



```python
# Load the input image into a numpy array and display it
input_image = caffe.io.load_image(IMAGE_FILE)
plt.imshow(input_image)
plt.show()

# Calculate how many 256x256 grid squares are in the image
rows = input_image.shape[0]/256
cols = input_image.shape[1]/256

# Initialize an empty array for the detections
detections = np.zeros((rows,cols))

# Iterate over each grid square using the model to make a class prediction
start = time.time()
for i in range(0,rows):
    for j in range(0,cols):
        grid_square = input_image[i*256:(i+1)*256,j*256:(j+1)*256]
        # subtract the mean image
        grid_square -= mean_image
        # make prediction
        prediction = net.predict([grid_square])
        detections[i,j] = prediction[0].argmax()
end = time.time()

# Display the predicted class for each grid square
plt.imshow(detections, interpolation=None)

# Display total time to perform inference
print 'Total inference time: ' + str(end-start) + ' seconds'
```

```
Total inference time: 0.761701107025 seconds
```