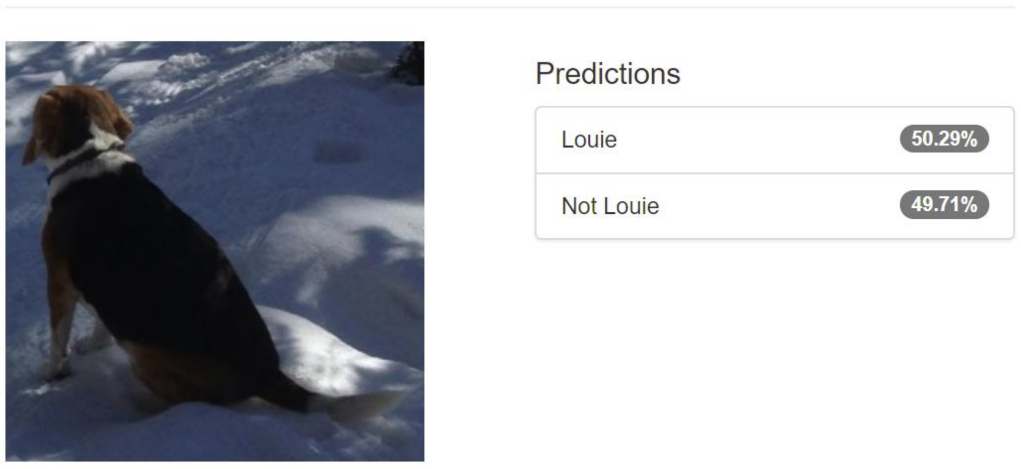
NVIDIA Fundamentals of Deep Learning with Computer Vision Certification Report

In this course, our workspace is in DIGITS.

Task1

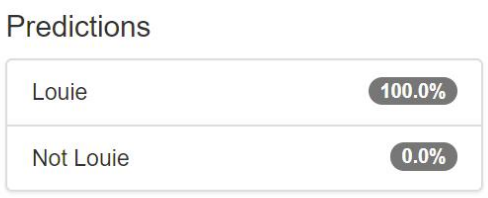
To begin with, we create a model: select "Images" -> "Classification" under "New Model".
After we create the model, we select our dataset: "Images of Beagles."
Next, we'll tell DIGITS how many times to look through the images. Each trip through the data is called an epoch. Start with 2 by changing the number of "Training epochs" from 30.
Then, we select the deep neural network to train by selecting "AlexNet" from the list of "Standard Networks".
After our model is trained a section titled, "Trained Models."
Under "Test a single image," there is a field titled "Image Path."
Copy and paste the following path: /dli/data/BeagleImages/Louie/louie1.JPG and select "Classify One."
The model then generates a prediction.



Predictions

Louie	50.29%
Not Louie	49.71%

Then, we create another model and set epochs = 100. Then we test our model:

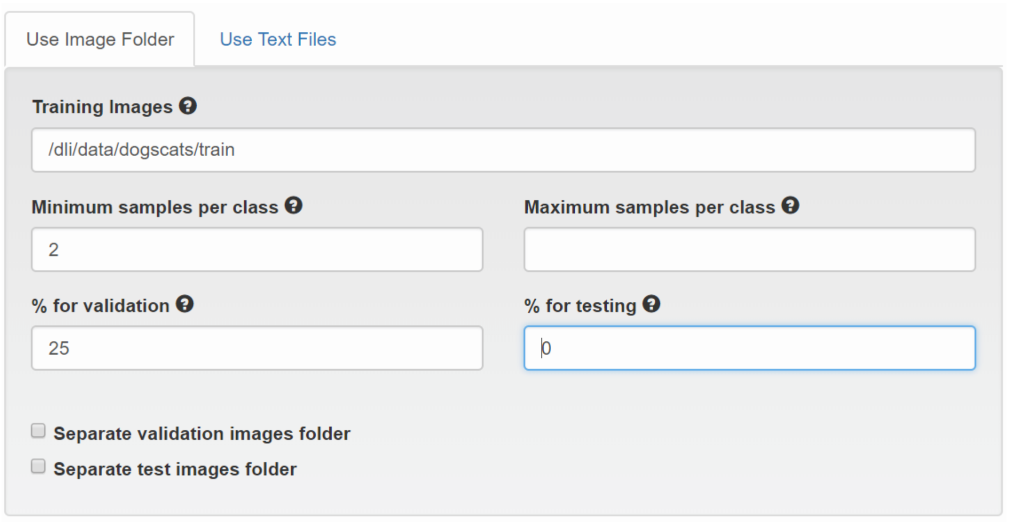

Predictions

Louie	100.0%
Not Louie	0.0%

We know that, if we want to get higher accuracy, we can enlarge epochs, which means multiple-training.

Task2

The purpose of this task is that we will have a trained neural network that can correctly classify dogs and cats that oure not part of the training dataset.
First, we create our model and select dataset:"/dli/data/dogscats/train".



Training Images: /dli/data/dogscats/train

Minimum samples per class: 2

Maximum samples per class:

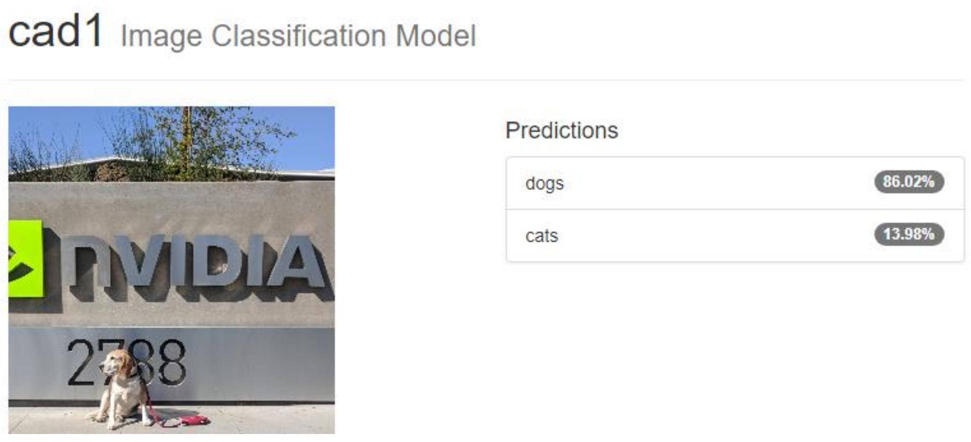
% for validation: 25

% for testing: 0

☐ Separate validation images folder

☐ Separate test images folder

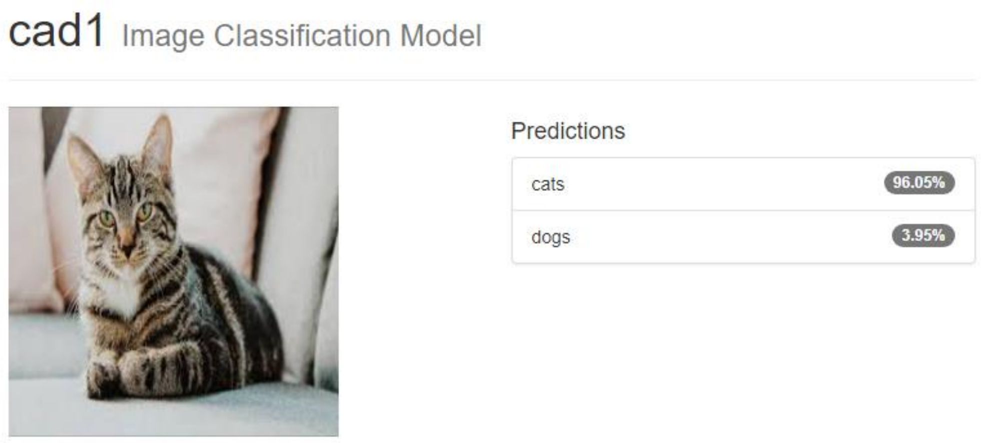
Here, we allocate 25% in the field % for validation.
Then we train that model.
After we train the model, we can test it like task1.



cad1 Image Classification Model

Predictions

dogs	86.02%
cats	13.98%



cad1 Image Classification Model

Predictions

cats	96.05%
dogs	3.95%

Task3

In this task, we make a trained network useful by removing it from its training environment and "deploying" it into an application.

```
MODEL_JOB_DIR = '/dli/data/digits/20180301-185638-e918'    ## Remember to set this to be the job directory for our model
!ls $MODEL_JOB_DIR
caffe_output.log     snapshot_iter_735.caffemodel     status.pickle
deploy.prototxt      snapshot_iter_735.solverstate    train_val.prototxt
original.prototxt    solver.prototxt
```

Again, our "model" consists of two files: the architecture and the weights.

The architecture is the file called deploy.prototxt and the weights are in the most recent snapshot file snapshot_iter_#.caffemodel.In this case, snapshot number 735 contains the weights learned after all 5 epochs.

```
ARCHITECTURE = MODEL_JOB_DIR + '/' + 'deploy.prototxt'
WEIGHTS = MODEL_JOB_DIR + '/' + 'snapshot_iter_735.caffemodel'
print ("Filepath to Architecture = " + ARCHITECTURE)
```

```
print("Filepath to weights = "+ WEIGHTS)
```

```
Filepath to Architecture = /dli/data/digits/20180301-185638-e918/deploy.prototxt
Filepath to weights = /dli/data/digits/20180301-185638-e918/snapshot_iter_735.caffemodel
```

Next, we need to make sure that the program that we're building can both read and process those files. For this basic type of deployment, we'll need to install (or include) the framework that they oure written in to be able to interpret them. We'll learn to deploy to environments that don't require installing the framework later in this course. We'll also need to use the GPU to take advantage of parallel processing. Again, our model consists of hundreds of thousands of operations that can be largely accelerated through parallelization.

```
import caffe
caffe.set_mode_gpu()
```

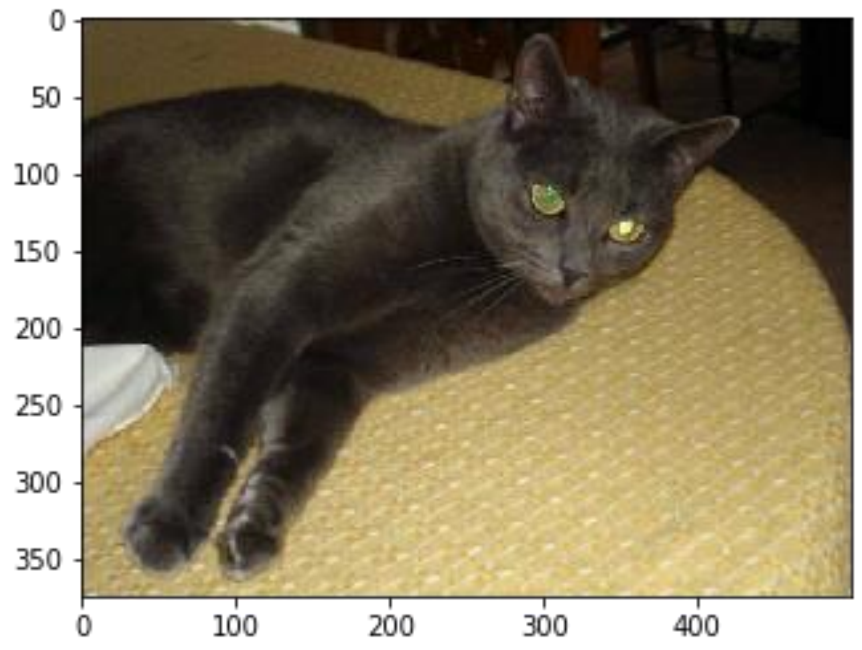
Next, we'll create a "Classifier" object called "net". The more common the workflow, the easier existing tools will make our project. In this case, image classification is very common, so this next code block simply takes our architecture file and weights file and a bit about the data and makes common actions easy.

```
# 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
```

The Classifier class includes a method called "predict", which takes an input of an image as defined above and generates an output of the likelihood of the image belonging to each category.

Creating an Expected Input: Preprocessing

```
import matplotlib.pyplot as plt #matplotlib.pyplot allows us to visualize results
input_image= caffe.io.load_image('/dli/data/dogscats/train/cats/cat.10941.jpg')
plt.imshow(input_image)
plt.show()
```

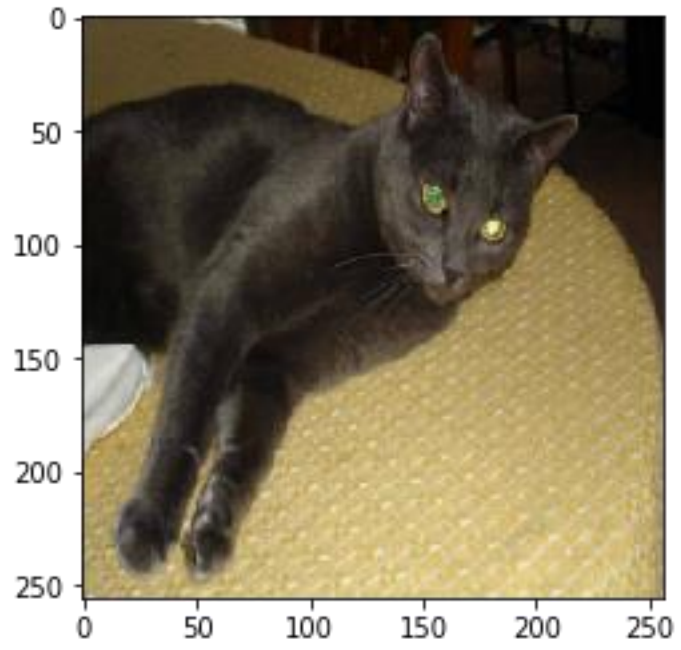


In this section, we'll examine the files generated when DIGITS created our dataset.

```
DATA_JOB_DIR = '/dli/data/digits/20180222-165843-ada0'    ## Remember to set this to be the job directory for our model
!ls $DATA_JOB_DIR
```

```
create_train_db.log   labels.txt           mean.jpg        train.txt   val.txt
create_val_db.log     mean.binaryproto     status.pickle   train_db    val_db
```

```
import cv2
input_image=cv2.resize(input_image, (256, 256), 0,0)
plt.imshow(input_image)
plt.show()
```



DIGITS normalized the images by subtracting the mean image from each image to reduce the computation necessary to train.

Load the mean image and subtract it from the test image below:

```
mean_image = caffe.io.load_image(DATA_JOB_DIR+'/mean.jpg')
ready_image = input_image-mean_image
```

Forward Propagation: Using our model

This is what we care about. Let's take a look at the function:

prediction = net.predict([grid_square]).

Like any function, net.predict passes an input, ready_image, 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.

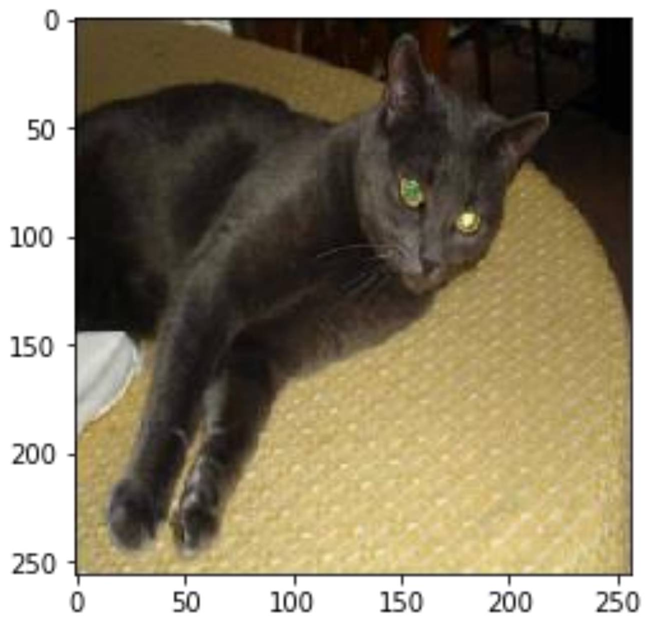
```
# make prediction
prediction = net.predict([ready_image])
print prediction
[[ 0.70993775    0.29006225]]
```

Generating a useful output: Postprocessing

At this point, we can really build whatever we want. Our only limit is our programming experience. Before getting creative, let's build something basic. This code will determine whether our network output a higher value for the likelihood of "dog" than it did for "cat." If so, it will display an image that would be appropriate if a dog approached our simulated doggy door. If not, the image represents what we'd want to happen if our network determined a cat was at the door.

```
print("Input image:")
plt.imshow(input_image)
plt.show()

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"
```

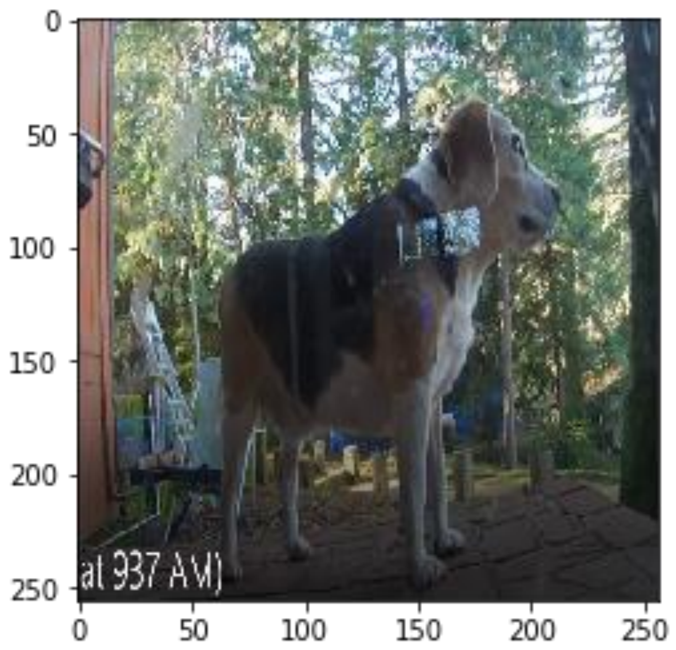
```
Input image:
```



```
Output:
Sorry cat:( https://media.giphy.com/media/jb8aFEQk3tADS/giphy.gif
```

```
##Create an input our network expects
input_image= caffe.io.load_image('/dli/data/fromnest.PNG')
input_image=cv2.resize(input_image, (256, 256), 0,0)
ready_image = input_image-mean_image
##Treat our network as a function that takes an input and generates an output
prediction = net.predict([ready_image])
print("Input Image:")
plt.imshow(input_image)
plt.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"
```

Input Image:



```
[[ 0.39924237    0.6007576 ]]
```

Output:

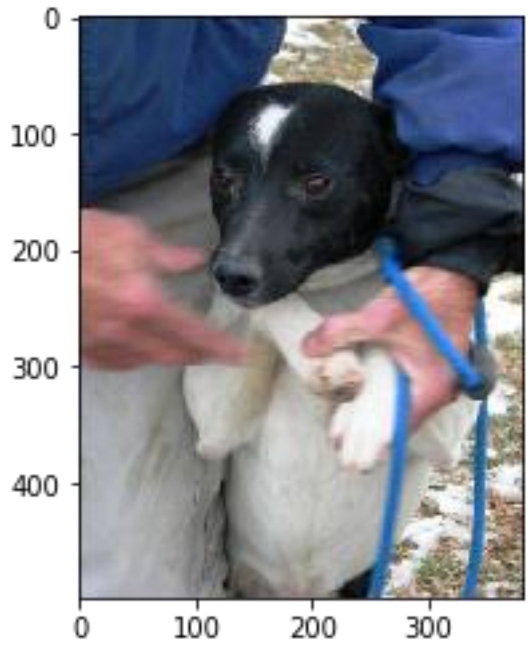
Welcome dog! https://www.flickr.com/photos/aidras/5379402670

Putting it all together

Let's put this deployment process together to see how it might look outside of this Jupyter notebook. In the Python file at pythondeployment.py, we'll see the same code as above, but consolidated into one file. We'll use this approach during our end of course assessment, so take a look. Insert the filepath to a test image here to visualize it.

pythondeployment.py:

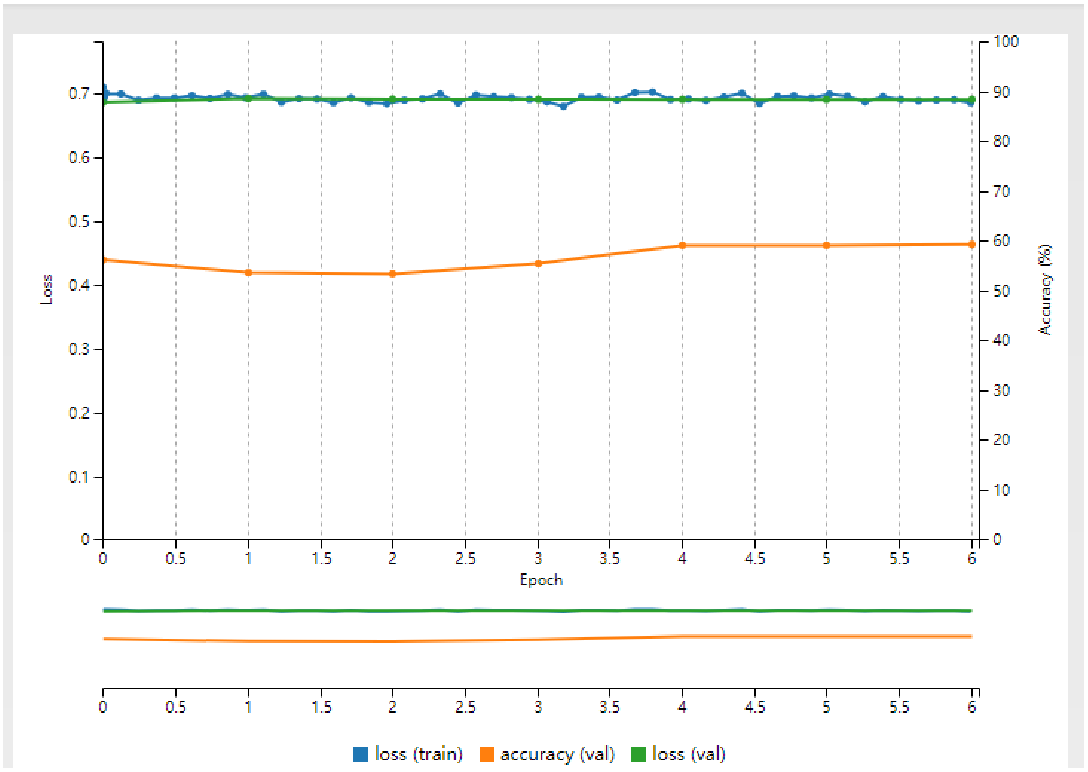
```
import caffe
import cv2
import sys
import matplotlib.pyplot as plt
#import Image

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])

    #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]))

TEST_IMAGE = '/dli/data/dogscats/test/1.jpg'
display= caffe.io.load_image(TEST_IMAGE)
plt.imshow(display)
plt.show()
```



```
!python pythondeployment.py $TEST_IMAGE 2>/dev/null
[[ 0.42844081    0.57155913]]
Output:
Welcome dog! https://www.flickr.com/photos/aidras/5379402670
None
```

Task4

We can now create a new model from this starting point. Go back to DIGITS' home screen by selecting "DIGITS" in the top left corner and create a new Image Classification model like before.
*New Model (Images) -> Classification *
Select the same dataset - (Dogs and Cats)
Choose some number of epochs between 3 and 8. (Note that in creating a model from scratch, this is where we could have requested more epochs originally.)
Change the learning rate to 0.0001
Select "advanced options" to "fix" the learning rate at that value
This time, instead of choosing a "Standard Network," select "Pretrained Networks."
Select the pretrained model that we just created, "Dogs vs. Cats".
Name our model - We chose "Study more"
Click Create
When creating the model, we'll get the following graph.



There are four categories of levers that we can manipulate to improve performance. Time spent learning about each of them will pay off in the performance of our models.

1) Data - A large and diverse enough dataset to represent the environment where our model should work. Data curation is an art form in itself.
2) Hyperparameters - Making changes to options like learning rate are like changing our training "style." Currently, finding the right hyperparameters is a manual process learned through experimentation. As we build intuition about what types of jobs respond well to what hyperparameters, our performance will increase.
3) Training time - More epochs improve performance to a point. At some point, too much training will result in overfitting (humans are guilty of this too), so this cannot be the only intervention we apply.
4) Network architecture - We'll begin to experiment with network architecture in the next section. This is listed as the last intervention to push back against a false myth that to engage in solving problems with deep learning, people need mastery of network architecture. This field is fascinating and powerful, and improving our skills is a study in math.
Instead of working through the deployment of the model you just retrained, it's time to introduce a shortcut to performance: deploying expert pretrained models.

In this section, you'll learn to deploy other people's networks so that you can get the performance gains of their research, compute time, and data curation.
Recall that the specific deep learning workflow this course started with is image classification. One reason we start with this task is because it is one of the most solved challenges in Deep Learning. It has benefited from the research community refining solutions to a competition called "ImageNet."

"ImageNet" is a large dataset with 1000 classes of common images. The competition granted awards for the research teams that had the lowest loss against this dataset. The network we have been working with, AlexNet, won Imagenet in 2012. Teams from Google and Microsoft have been winners since then.
Here's the exciting part. 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, we can deploy award winning neural networks.
All we need to deploy one of these models are the model's architecture and weights. A quick Google search for "pretrained model alexnet imagenet caffe" returns multiple pages to download this model.
We'll download them both using a tool called wget. 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.

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

```
--2019-02-13 00:21:26--  http://dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel
Resolving dl.caffe.berkeleyvision.org (dl.caffe.berkeleyvision.org)... 169.229.222.251
Connecting to dl.caffe.berkeleyvision.org (dl.caffe.berkeleyvision.org)|169.229.222.251|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 243862414 (233M) [application/octet-stream]
Saving to: ‘bvlc_alexnet.caffemodel’

bvlc_alexnet.caffem 100%[===================>] 232.56M  13.3MB/s    in 11s

2019-02-13 00:21:38 (20.2 MB/s) - ‘bvlc_alexnet.caffemodel’ saved [243862414/243862414]

--2019-02-13                                                                 00:21:38--
https://raw.githubusercontent.com/BVLC/caffe/master/models/bvlc_alexnet/deploy.prototxt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.200.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.200.133|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 3629 (3.5K) [text/plain]
Saving to: ‘deploy.prototxt’

deploy.prototxt     100%[===================>]   3.54K  --.-KB/s    in 0s

2019-02-13 00:21:38 (76.2 MB/s) - ‘deploy.prototxt’ saved [3629/3629]
```

Those are the same two files that DIGITS generated when we trained a model from scratch. The only other file we took from DIGITS was the mean image that was used during training. We can download that below.

```
!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
```

```
--2019-02-13                                                                    00:23:48--
https://github.com/BVLC/caffe/blob/master/python/caffe/imagenet/ilsvrc_2012_mean.npy?raw
=true
Resolving github.com (github.com)... 192.30.253.113, 192.30.253.112
Connecting to github.com (github.com)|192.30.253.113|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location:
https://github.com/BVLC/caffe/raw/master/python/caffe/imagenet/ilsvrc_2012_mean.npy
[following]
--2019-02-13                                                                    00:23:48--
https://github.com/BVLC/caffe/raw/master/python/caffe/imagenet/ilsvrc_2012_mean.npy
Reusing existing connection to github.com:443.
HTTP request sent, awaiting response... 302 Found
Location:
https://raw.githubusercontent.com/BVLC/caffe/master/python/caffe/imagenet/ilsvrc_2012_mea
n.npy [following]
--2019-02-13                                                                    00:23:48--
https://raw.githubusercontent.com/BVLC/caffe/master/python/caffe/imagenet/ilsvrc_2012_mea
n.npy
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.200.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.200.133|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 1572944 (1.5M) [application/octet-stream]
Saving to: ‘ilsvrc_2012_mean.npy?raw=true’

ilsvrc_2012_mean.np 100%[===================>]    1.50M   --.-KB/s      in 0.07s

2019-02-13 00:23:48 (21.7 MB/s) - ‘ilsvrc_2012_mean.npy?raw=true’ saved [1572944/1572944]
```

Those files are now all stored on this server as:

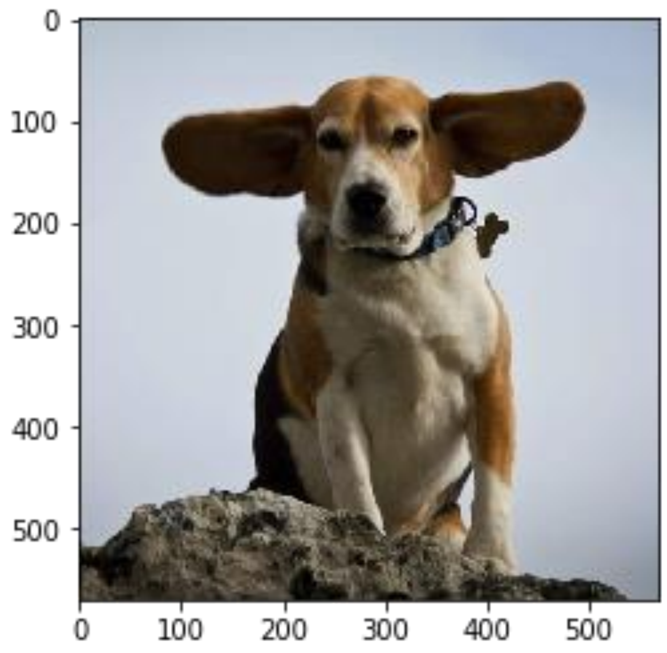
```
/dli/tasks/task4/task/deploy.prototxt
/dli/tasks/task4/task/bvlc_alexnet.caffemodel
/dli/tasks/task4/task/ilsvrc_2012_mean.npy
```

Start by initializing the model:

```
import caffe
import numpy as np
caffe.set_mode_gpu()
import matplotlib.pyplot as plt #matplotlib.pyplot allows us to visualize results

ARCHITECTURE = 'deploy.prototxt'
WEIGHTS = 'bvlc_alexnet.caffemodel'
MEAN_IMAGE = 'ilsvrc_2012_mean.npy'
TEST_IMAGE = '/dli/data/BeagleImages/louietest2.JPG'

# Initialize the Caffe model using the model trained in DIGITS
net = caffe.Classifier(ARCHITECTURE, WEIGHTS) #Each "channel" of our images are 256 x 256
```

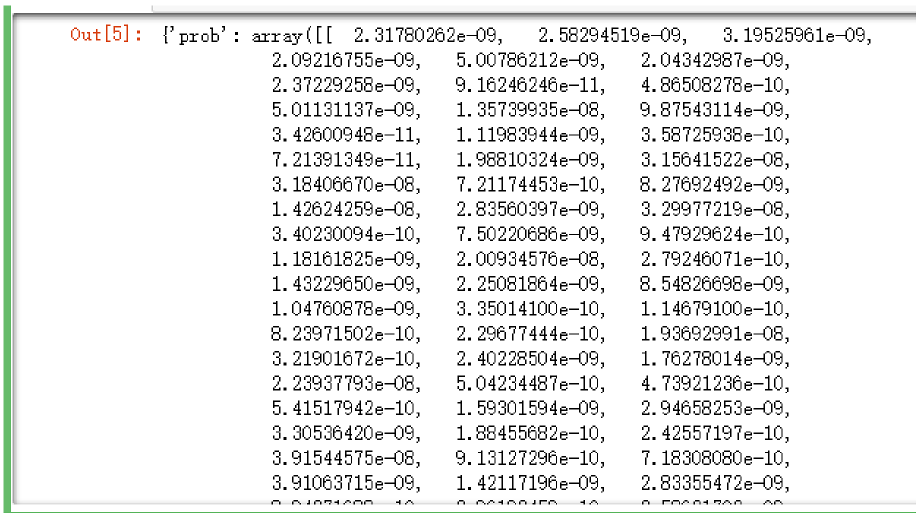
Then create an input the network expects. Note that this is different than the preprocessing used in the last model. To learn how imagenet was preprocessed, the documentation was clearly presented on the Caffe website:

```
#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, e.g., for different batch sizes)
net.blobs['data'].reshape(1,               # batch size
                          3,               # 3-channel (BGR) images
                          227, 227)    # image size is 227x227

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



Run the function and visualize the output.

```
# copy the image data into the memory allocated for the net
net.blobs['data'].data[...] = transformed_image

### perform classification
output = net.forward()

output
```



```
Out[5]: {'prob': array([[  2.31780262e-09,   2.58294519e-09,   3.19525961e-09,
                  2.09216755e-09,   5.00786212e-09,   2.04342987e-09,
                  2.37229258e-09,   9.16246246e-11,   4.86508278e-10,
                  5.01131137e-09,   1.35739935e-08,   9.87543114e-09,
                  3.42600948e-11,   1.11983944e-09,   3.58725938e-10,
                  7.21391349e-11,   1.98810324e-09,   3.15641522e-08,
                  3.18406670e-08,   7.21174453e-10,   8.27692492e-09,
                  1.42624259e-08,   2.83560397e-09,   3.29977219e-08,
                  3.40230094e-10,   7.50220686e-09,   9.47929624e-10,
                  1.18161825e-09,   2.00934576e-08,   2.79246071e-10,
                  1.43229650e-09,   2.25081864e-09,   8.54826698e-09,
                  1.04760878e-09,   3.35014100e-10,   1.14679100e-10,
                  8.23971502e-10,   2.29677444e-10,   1.93692991e-08,
                  3.21901672e-10,   2.40228504e-09,   1.76278014e-09,
                  2.23937793e-08,   5.04234487e-10,   4.73921236e-10,
                  5.41517942e-10,   1.59301594e-09,   2.94658253e-09,
                  3.30536420e-09,   1.88455682e-10,   2.42557197e-10,
                  3.91544575e-08,   9.13127296e-10,   7.18308080e-10,
                  3.91063715e-09,   1.42117196e-09,   2.83355472e-09,
```

What you see above is an array containing the probabilities that our image belongs to each of 1000 classes. Let's work to make this useful.

```
output_prob = output['prob'][0]   # the output probability vector for the first image in the batch
print 'predicted class is:', output_prob.argmax()
```

```
predicted class is: 162
```

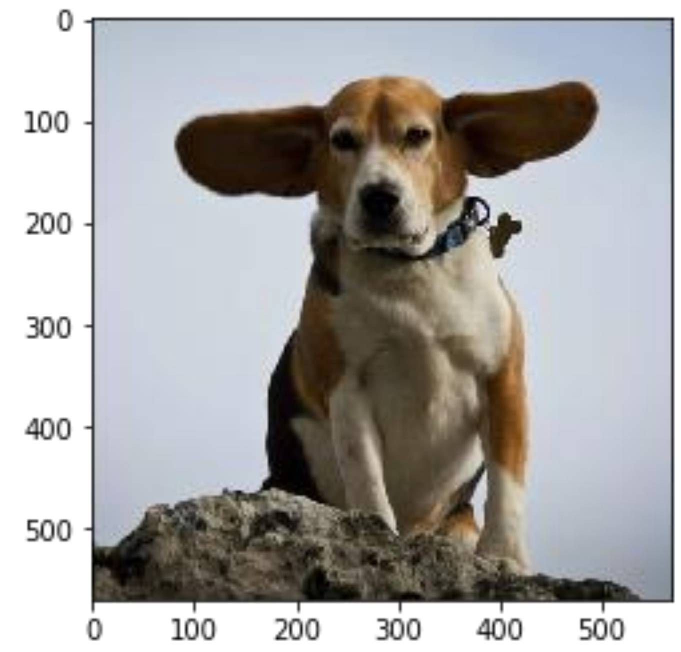
Using wget to get a dictionary (dict) of class to label.

```
!wget
https://raw.githubusercontent.com/HoldenCaulfieldRye/caffe/master/data/ilsvrc12/synset_word
s.txt
labels_file = 'synset_words.txt'
labels = np.loadtxt(labels_file, str, delimiter='\t')

print 'output label:', labels[output_prob.argmax()]
```

```
--2019-02-13                                                                    00:29:40--
https://raw.githubusercontent.com/HoldenCaulfieldRye/caffe/master/data/ilsvrc12/synset_word
s.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.200.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.200.133|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 31675 (31K) [text/plain]
Saving to: ‘synset_words.txt’

synset_words.txt        100%[===================>]    30.93K    --.-KB/s        in 0.01s

2019-02-13 00:29:40 (2.75 MB/s) - ‘synset_words.txt’ saved [31675/31675]

output label: n02088364 beagle
```

To get a clean view of what our application does, here is the input and output of our application.

```
print ("Input image:")
plt.imshow(image)
plt.show()

print("Output label:" + labels[output_prob.argmax()])
```



Output label:n02088364 beagle

Task5

Let's start by instantiating our model in the same way.

```
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 dataset

MODEL_FILE = MODEL_JOB_DIR + '/deploy.prototxt'                  # This file contains the description of the network architecture
PRETRAINED = MODEL_JOB_DIR + '/snapshot_iter_735.caffemodel'     # This file contains the *weights* that were "learned" during training
MEAN_IMAGE = DATASET_JOB_DIR + '/mean.jpg'                       # This file contains the mean image of the entire dataset. Used to preprocess the data.

# 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
```

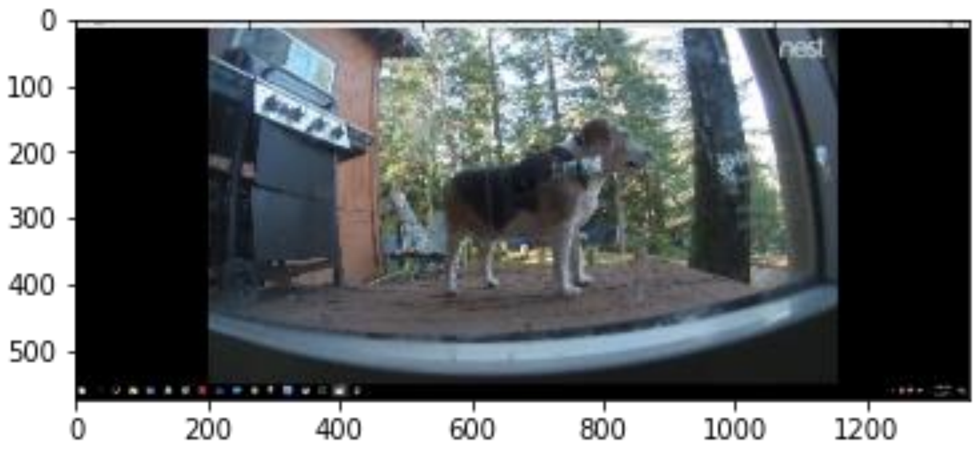
deploy models in series.

```
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.")
```

```
Ready to predict.
```

Next, we'll take an image directly from a front door security camera. Note, this image is larger than 256X256. We want to know where a dog is in this image.

```
# 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()
```



Run the cell below to see the prediction from the top left 256X256 grid_square of our random image.

```
X = 0
Y = 0

grid_square = input_image[X*256:(X+1)*256,Y*256:(Y+1)*256]
# subtract the mean image (because we subtracted it while we trained the network - more on this in next lab)
grid_square -= mean_image
# make prediction
```

```
prediction = net.predict([grid_square])
print prediction
```

```
[[ 0.89857852    0.10142148]]
```

This is the output of the last layer of the network we're using.

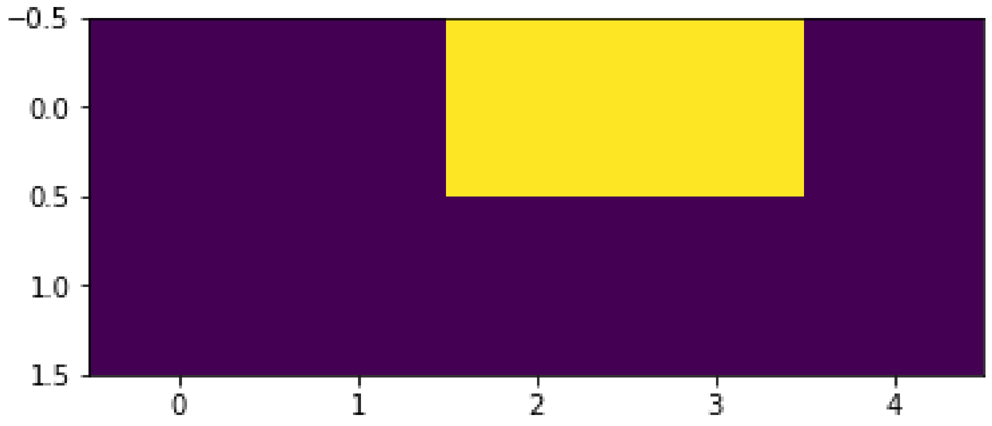
Next, let's implement some code around our function to iterate over the image and classify each grid_square to create a heatmap. Note that the key lesson of this section is that we could have built ANYTHING around this function, limited only by your creativity.

Depending on your programming (specifically Python) experience, you may get very different information out of the following cell. At its core, this block is cutting our input image into 256X256 squares, running each of them through our dog classifier, and creating a new image that is blue where there is not a dog and red where there is one, as determined by our classifier.

```
# 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: 0.805495023727 seconds
```



Approach 2 - Rebuilding from an existing neural network

To look at AlexNet's Caffe description in DIGITS:

1) Select the model "Dogs vs. Cats".

2) Select "Clone Job" (on the top right in blue) to copy all of the settings to a new network. Cloning networks is an important way to iterate as you experiment.

3) At the bottom of the model settings, where you selected AlexNet, select "Customize."

You'll see some prototext. That is the Caffe description of the AlexNet network.

Rule 1: Data must be able to flow

We removed the fc6 layer, but it's clear that it's still present when we visualize our network. This is because there are other layers that reference fc6:

fc7

drop6

relu6

Take a look at the Caffe prototext to see where those layers are told to connect to fc6 and connect them instead to your new layer, conv6.

Let's take a second to compare the layer we removed with the layer we added.

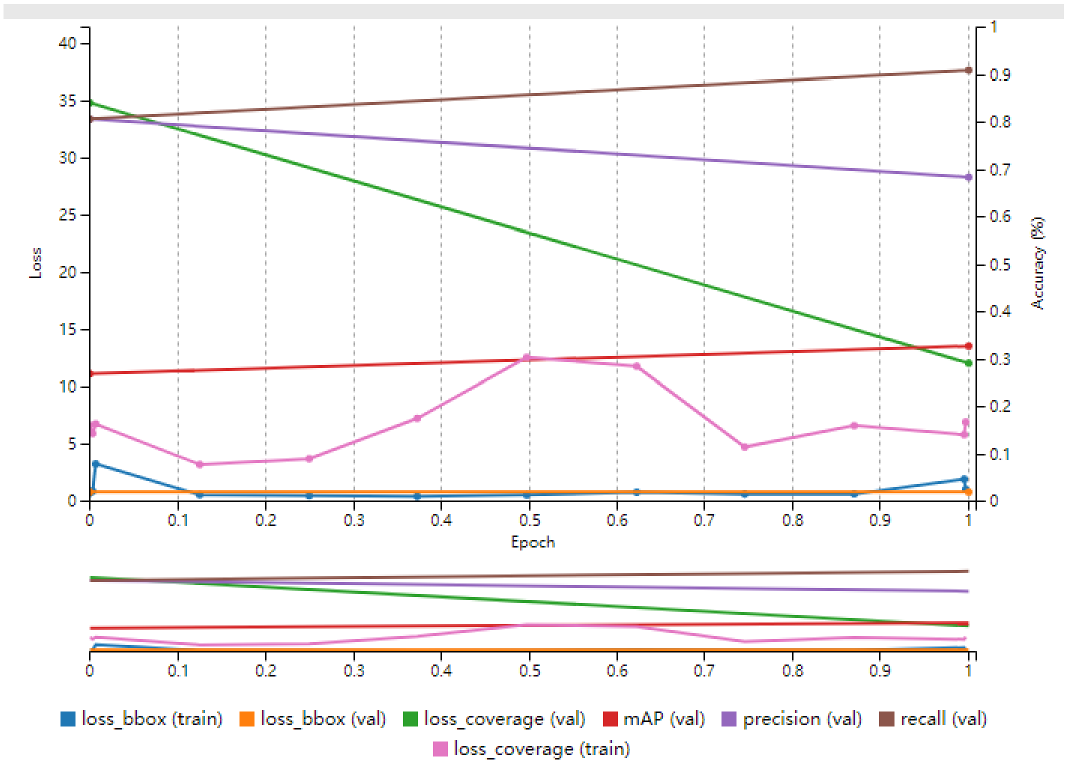
We removed a "fully connected" layer, which is a traditional matrix multiplication. While there is a lot to know about matrix multiplication, the one characteristic that we'll address is that matrix multiplication only works with specific matrix sizes. The implications of this for our problem is that

we are limited to fixed size images as our input (in our case 256X256).
We added a "convolutional" layer, which is a "filter" function that moves over an input matrix. There is also a lot to know about convolutions, but the one characteristic that we'll take advantage of is that they do not only work with specific matrix sizes. The implications of this for our problem are that if we replace all of our fully connected layers with convolutional layers, we can accept any size image.
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.
Let's convert AlexNet into a Fully-Convolutional Network (somewhat confusingly referred to as an FCN). Replace fc7 through fc8 with equivalent convolutional layers using the following Caffe text.
Approach 3: DetectNet
To build an end-to-end supervised deep learning workflow, we'll need labeled data. Until this point, we've defined "labeled" as an indicator of which category each image belonged to. However, the real task of labeling data is **curating input output mappings."
In this case, we want our inputs to be an image of ANY size and our outputs to indicate the location of objects within the image. First, an input image:
input_image = caffe.io.load_image('/dli/data/train/images/488156.jpg') #!ls this directory to see what other images and labels you could test
plt.imshow(input_image)
plt.show()

Next, its corresponding label:
!cat '/dli/data/train/labels/488156.txt' #"cat" has nothing to do with the animals, this displays the text
dog 0 0 0 161.29 129.03 291.61 428.39 0 0 0 0 0 0 0
surfboard 0 0 0 31.25 36.44 410.41 640.0 0 0 0 0 0 0 0

When the "New Object Detection Dataset" panel opens, use the preprocessing options from the image below (Watch for leading/training spaces):

Training image folder: /dli/data/train/images
Training label folder: /dli/data/train/labels
Validation image folder: /dli/data/val/images
Validation label folder: /dli/data/val/labels/dog
Pad image (Width x Height): 640 x 640
Custom classes: dontcare, dog
Group Name: MS-COCO
Dataset Name: coco-dog
New model:



We learn how to train multiple types of neural networks using DIGITS
What types of changes you can make to how you train networks to improve performance
How to combine deep learning with traditional programming to solve new problems
How to modify the internal layers of a network to change functionality and limitations
To see who else has worked on problems like yours and what parts of their solution you can use