

Report for NVIDIA Fundamentals of Deep Learning with Computer Vision

Task 1

For the first task, we encounter the problem about image classification. we train a neural network to separate images into classes. The first task is an easy one to tackle with. We just need recognize Louie's image from other dogs' images, thus there are two classes we should distinguish in this case: Louie and Not Louie.

It's a simple dataset which contains a few images for me to handle with. Nvidia DIGITS is a powerful tool which can be convenient for novice in code and deep learning to manipulate. DIGITS the tool simplifies common deep learning tasks such as managing data, designing and training neural networks on multi-GPU systems, monitoring performance in real time with advanced visualizations, and selecting the best performing model from the results browser for deployment. DIGITS is completely interactive so that data scientists can focus on designing and training networks rather than programming and debugging.

And the Result for the model is pretty well after 100 epochs although the volume of training dataset is small:

Louie Classifier after 100 Epochs Image Classification Model



Predictions	
Louie	100.0%
Not Louie	0.0%

Louie Classifier after 100 Epochs Image Classification Model

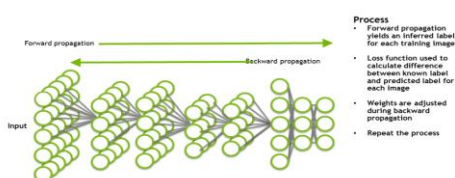


Predictions	
Not Louie	99.92%
Louie	0.08%

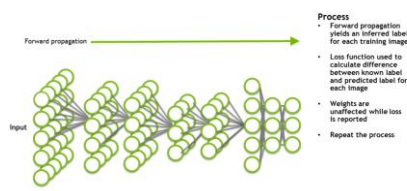
Task 2

For the second task, we will train a neural network to perform well on new data which includes a bunch of data for our training and testing datasets. Since we use these datasets, so that we can totally experience the whole process of the training and validating of deep learning. Here are diagrams of approaches for the two main method provided by task:

DEEP LEARNING APPROACH - TRAINING

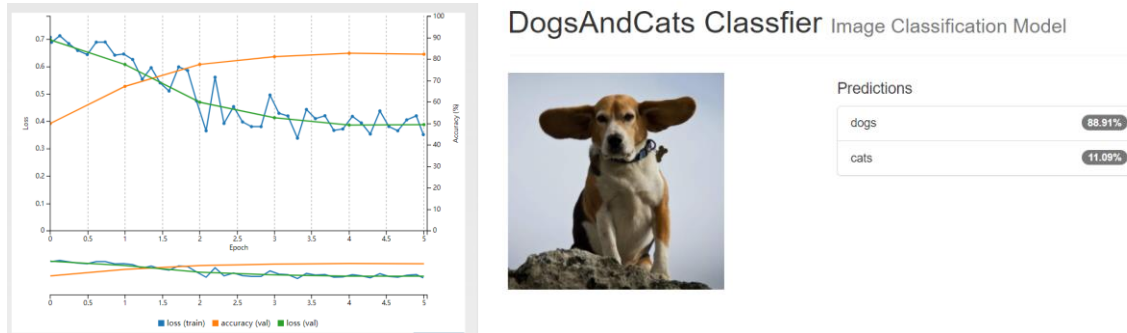


DEEP LEARNING APPROACH - VALIDATION



As the diagram showed, the major different between the training approach and validating or tasting approach of deep learning is whether it exits a backward propagation, so the images in testing dataset can always be used as a brand new one for machine to assess accuracy of the model without being remembered by the model itself.

In the case of this task, our model can figure out new images which the model haven't seen before from validate dataset, and via the training proceed, it can definitely recognize whether a image contains a dog or a cat. Following the instruct of task 2, we finally complete a plot about accuracy of the model we set. Meanwhile we tested a new image and the model gave us a result seems well.



Task3

With the function in model I have created from last task, in this task, I exerted how to use this model in the specific situation such as application or solution. I experience the course of deploying my model. Through the task, I can implement my model to some practical problem with right code to deploy the mode.

Especially, in this task part, we used three lines code to load us model to the application. The point which should be obviously noticed is file path of our two augments: Architecture and Weight. They are both vital for the success of deploying model correctly.

```
MODEL_JOB_DIR = '/dli/data/digits/20180301-185638-e918' ## Remember to set this to be the job directory for your 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
```

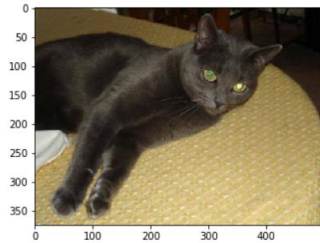
```
ARCHITECTURE = MODEL_JOB_DIR + '/' + 'deploy.prototxt'
WEIGHTS = MODEL_JOB_DIR + '/' + 'snapshot_iter_735.caffemodel'
print("Filepath to Architecture = " + ARCHITECTURE)
print("Filepath to weights = " + WEIGHTS)

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

```
# 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
```

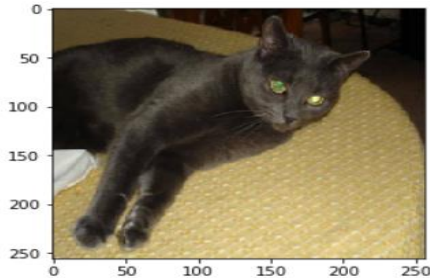
Another important issue for our model to run well is creating for an expected input, the input can be accepted by the model configuration and suitable in size.

```
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()
```



For the normalized image, hence we resize the input first to like this:

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



Then we use mean image of the dataset to subtract to each image so that we could reduce the computation:

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

And after all the input normalized and edited, we can use the model in the application to predict:

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

[[ 0.70993775  0.29006225]]
```

We can't easily distinguish these two number in output directly produced by the model, so we modified it by some simple method to show more obvious result that we can read:

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

print("Output:")
if prediction.argmax() == 0:
    print("Sorry cat! https://media.alfor.com/media/3b8aFEQ3tAD5/alphor.gif")
else:
    print("Welcome dog! https://www.flickr.com/photos/aidras/5379402670")

Input Image:
Output:
Sorry cat! https://media.alfor.com/media/3b8aFEQ3tAD5/alphor.gif
```




At this time, we get the whole sight of application at end, it is convenient for us to handle any type of image caught by the door with a nearly suitable answer:

```

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>

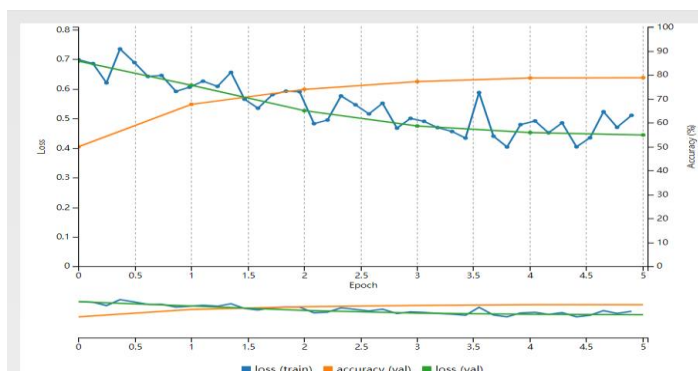
In conclude, deploying is key component for a project of deep learning to work in practical world situation, and what we want are just about we can input the acceptable data to model and output expectable data to user.

Task4

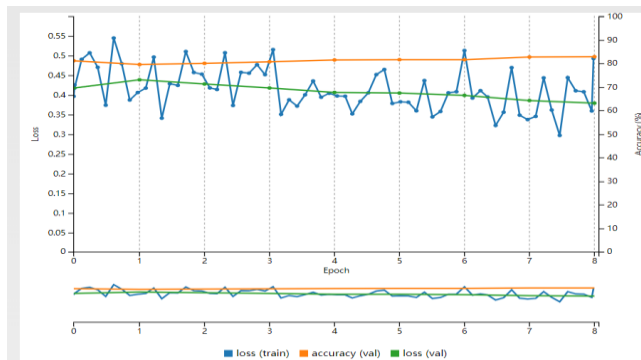
In the task, we hold on an experiment to improve the performance of our model by several different method in every steps of whole training process. As the course says, there are three ways for us to optimize:

- 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

Back to our original five epochs model, the performance of it displayed like this:



Through the guidance of the notebook, we change the number of epoch to 8 with learning rate changed to 0.0001 , and it's performance showing get a huge alter:



We could comment to the result above: As expected, the accuracy starts close to where our first model left off, 80%. Accuracy does continue to increase, showing that increasing the number of epochs often does increase performance. The rate of increase in accuracy slows down, showing that more trips through the same data can't be the only way to increase performance. We should try different combination of four parameters in this model to modify and elevate quality includes data, hyperparameters, training time and network architecture.

It's also a good way for us to deploy award winning models which are much better by experts taking bunch of time to ameliorate. In this case, we use ImageNet which 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.

By the powerful tool wget we can download data directly from the server and we use the mean image we already have with data we get. After that, we initiate 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 = 'ilsvrc2012_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 we an input the network expects:

```
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 value

# 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]
# 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
net.blobs['data'].reshape(1, (3,1,1)) # reshape channels from RGB to BGR
# batch size
1,
227, 227) # image size is 227x227

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



As same as we did before, we should make a output useful to user:

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
```

This is closer. Take a look at imagenet's classes to see what that number corresponds to [here](#). Better???? Let's add that functionality to our application so we have a useful end-to-end deployment.

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

```
# wget https://raw.githubusercontent.com/HoldenCaulfieldRye/caffe/master/data/ilsrvcl2/synset_words.txt
labels_file = 'synset_words.txt'
labels = np.loadtxt(labels_file, str, delimiter='\t')

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

--2019-02-22 17:44:37-- https://raw.githubusercontent.com/HoldenCaulfieldRye/caffe/master/data/ilsrvcl2/synset_words.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

2019-02-22 17:44:37 (2.90 MB/s) - 'synset_words.txt' saved [31675/31675]

output label: n02088364 beagle
```

And we successfully deployed a award model for our application to use. The clear view of input and output displayed as this:

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

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

Input image:

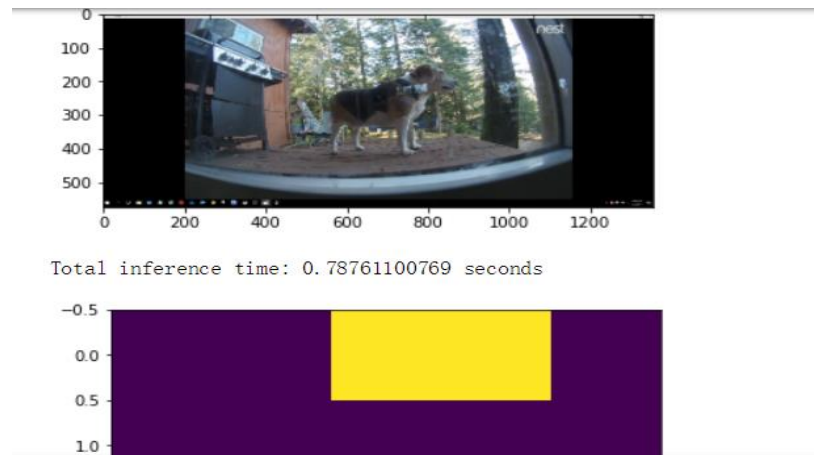
Output label:n02088364 beagle
```

In this part of task, we have learn to deploy other people's networks so that you can get the performance gains of their research, compute time, and data curation. Using these resource we can even edit and optimize these resource and create multiple wider way for the application to run a much more perfect performance.

Task 5

In this task, our main target is object detection which is another common workflow. And in this point in our exploration of deep learning to try a different workflow. Our first method of solving an object detection problem will be to combine an image classification network with traditional programming to create the input and output pairing that we want. And the traditional programming for object detection that We're going to is sliding window and for explanation and definition in notebook: "we'll take split out image into small sections which we'll call grid squares. We're run each grid square through an image classifier. If that grid square contains an image of a dog, we'll have localized Louie in the image. "

The result for traditional method sliding window:



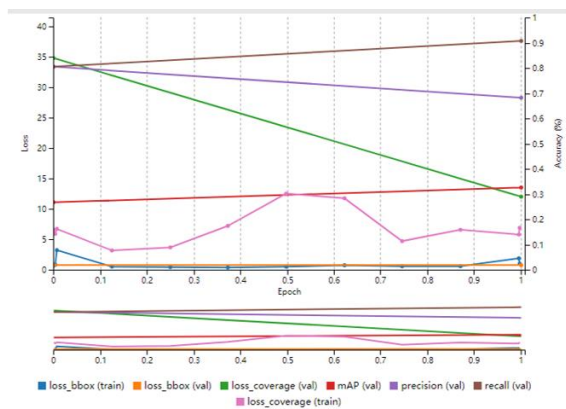
There are a question in notebook about some factors we didn't think about when building this solution and how we could account for them and for the answer: The fact that we used a sliding window with non-overlapping grid squares means that it is very likely that some of our grid squares will only partially contain a dog and this can lead to misclassifications. However, as we increase the overlap in the grid squares we will rapidly increase the computation time for this sliding window approach. We can make up for this increase in computation by *batching* inputs, a strategy that takes advantage of the GPU's intrinsic strength of parallel processing.

And now so on we try the second approach: Rebuilding from an existing neural network. We use the AlexNet we always embedder our model before. But we enter the core of the network structure to check details about each layer realizing code, and we use a convolutional layer to replace a full connected layer of the AlexNet and it structure in code list:

```
layer {
  name: "conv6"
  type: "Convolution"
  bottom: "pool5"
  top: "conv6"
  param {
    lr_mult: 1.0
    decay_mult: 1.0
  }
  param {
    lr_mult: 2.0
    decay_mult: 0.0
  }
  convolution_param {
    num_output: 4096
    pad: 0
    kernel_size: 6
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
```


And we continue replacing all the fully connected layers with convolutional layers and get the new network structure base on AlexNet. It's performance in detection is much better.

And the third approach is that we could use a special DetectNet for this problem. We set the corresponding dataset and use the dataset to our customized model. We copy a really long code with a lot of layer setting in it and we can finally get performance of the model:



And with the powerful tool DIGITS, we can customized many different models easily and matches various of dataset to fix the specific problem.