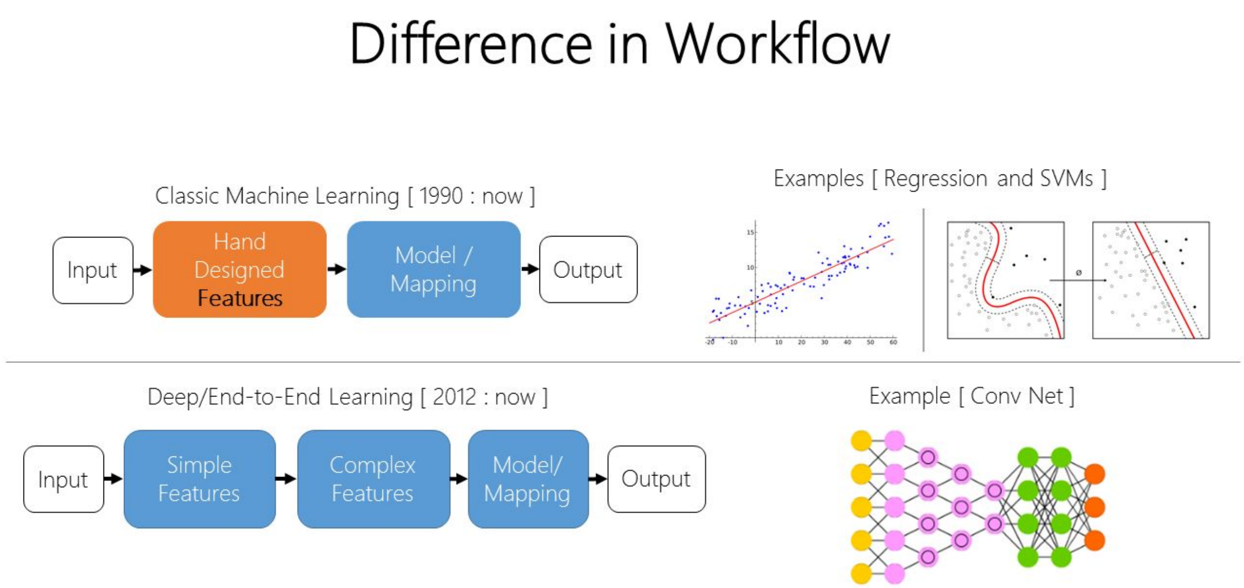
Deep Learning in Computer Vision

Chapter 0 : Concept of Deep Learning

Deep learning allows computers to "learn" from examples (data). Solving problems with deep learning requires identifying some pattern in the world, finding examples that highlight both sides of the pattern (the input and the output), and then letting a "neural network" learn the map between the two. This opens the types of problems where computers can help us to those where we:

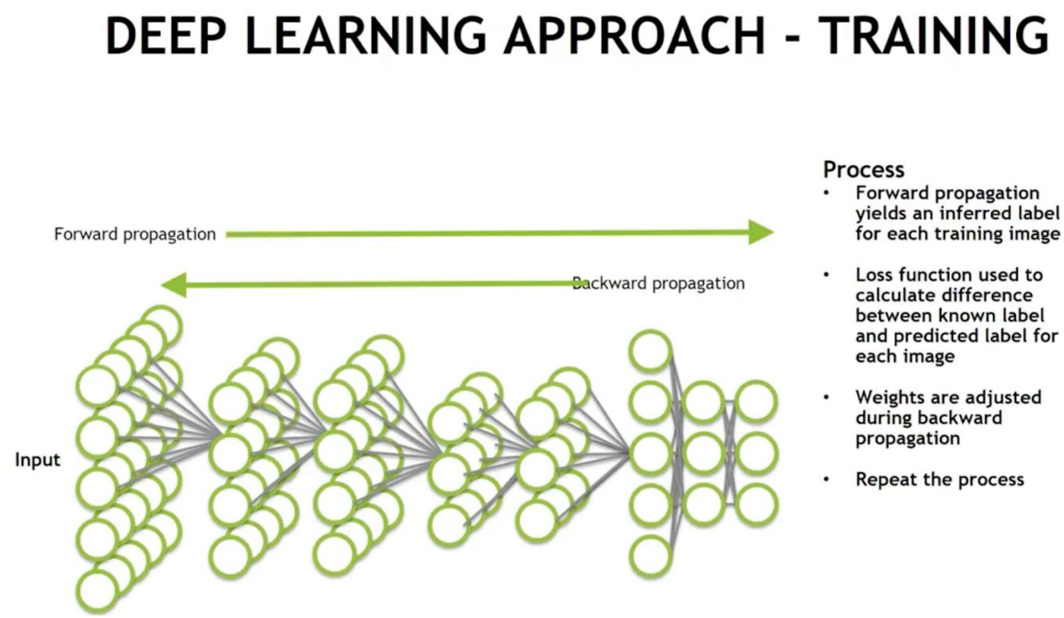
1. Have identified a pattern within a problem
2. Have enough data that exemplifies the pattern



Conclusion: Deep Learning removes the need of writing explicit instructions.

Chapter 1 : Training Deep Learning Networks

Concepts:
Deep Neural Networks are flexible algorithms inspired by the human brain that allow practitioners to use training strategies inspired by human learning. The input of an image generated an output of the network's confidence that the image belonged to one of two classes.



Experiments:
Deep Neural Networks: GPU Task 1

Using a labeled dataset to train a network to predict whether the picture is louis or not. After one epoch, our model was predicting no better than chance: 50/50.

Predictions

Louie	50.29%
Not Louie	49.71%

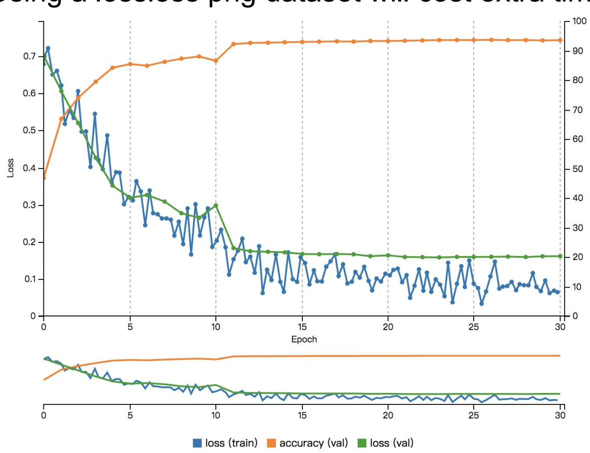
But after 100 epochs of training, the model performed well.

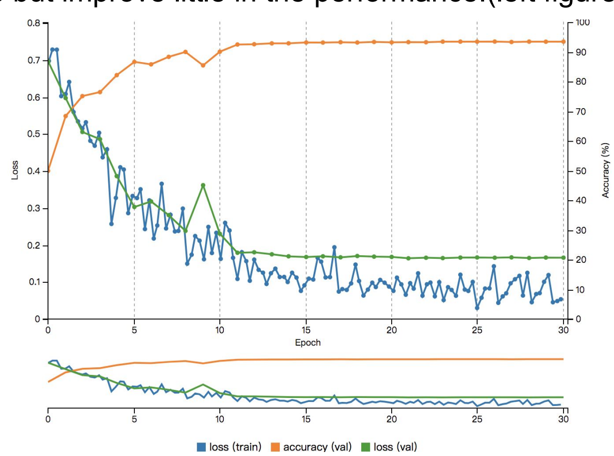
Chapter 2: Big data

Concepts:
In the era of big data, we create ~2.5 quintillion bytes of data per day. Free datasets are available from places like Kaggle.com and UCI. Crowdsourced datasets are built through creative approaches - e.g. Facebook asking users to "tag" friends in their photos to create labeled facial recognition datasets. More complex datasets are generated manually by experts - e.g. asking radiologists to label specific parts of the heart.

Experiment:

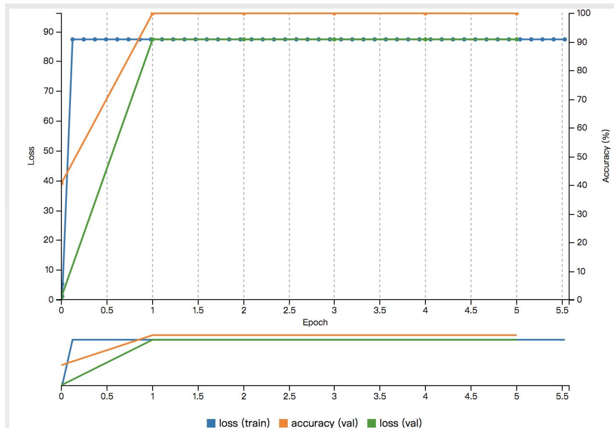
In this part we tried different configurations in the network.
Using a lossless png dataset will cost extra time but improve little in the performance.(left figure)



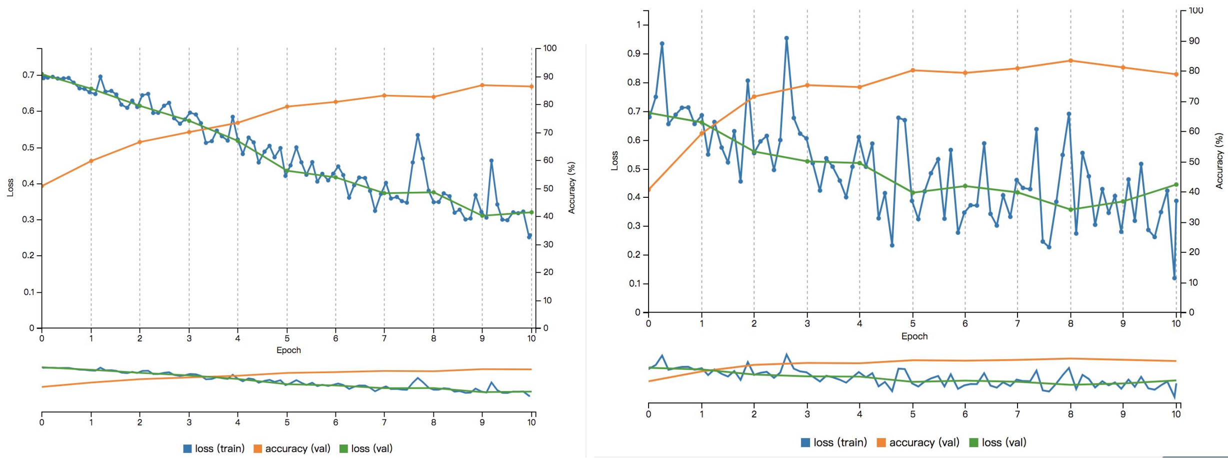


On the contrary, Using a dataset without any encoding will not reduce the performance but save a lot of time. (right figure)

Trying to use other networks showing similar results, but different learning curves.
LeNet:

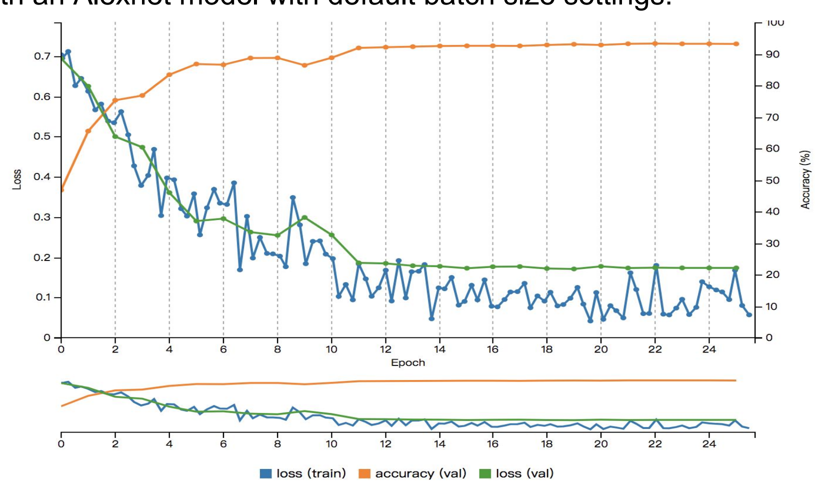


Trying out different batch sizes with a fixed learning rate at 0.01:



batch size of: 512(left figure), and 32(right figure)

In comparison with an Alexnet model with default batch size settings:



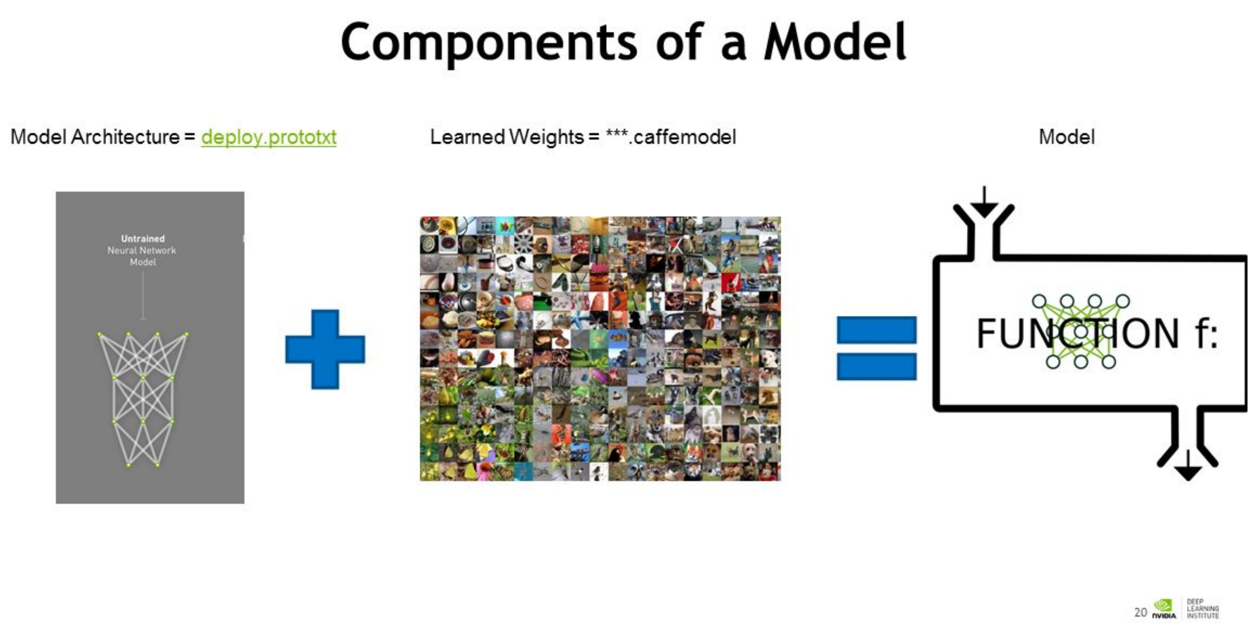
Conclusion: Use dynamic learning rate and batch sizes according to the requirement.

Chapter 3: Deploying our Model

Concept:

A trained network consists of two components:

1. A description of the architecture of the untrained network
2. The weights that were "learned" while the network trained



We can deploy the model into various of problems.

Experiment:

We can see the model elements saved in DIGIT. Use them well, in the caffe model.

```
MODEL_JOB_DIR = '/dli/data/digits/20180301-185638-e918'  ## Remember to set
!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
```

```
# Initialize the Caffe model using the model trained in DIGITS
net = caffe.Classifier(ARCHITECTURE, WEIGHTS,
                       channel_swap =(2, 1, 0), #Color images h
                       raw_scale=255) #Each pixel value is a nu
```

And thus we can make predictions.

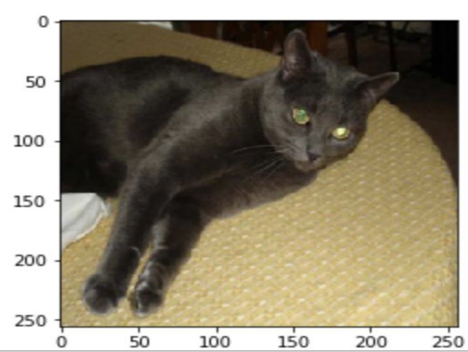
```
# make prediction
prediction = net.predict([ready_image])
print prediction
```

```
[[ 0.70993775  0.29006225]]
```

After predictions, there can be still a post-processing, that is interpret and use the prediction for certain purposes. For example…...

```
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:
```



Chapter 4

Concept: This chapter focusing on improving the training performance based on our pretrained model above.

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

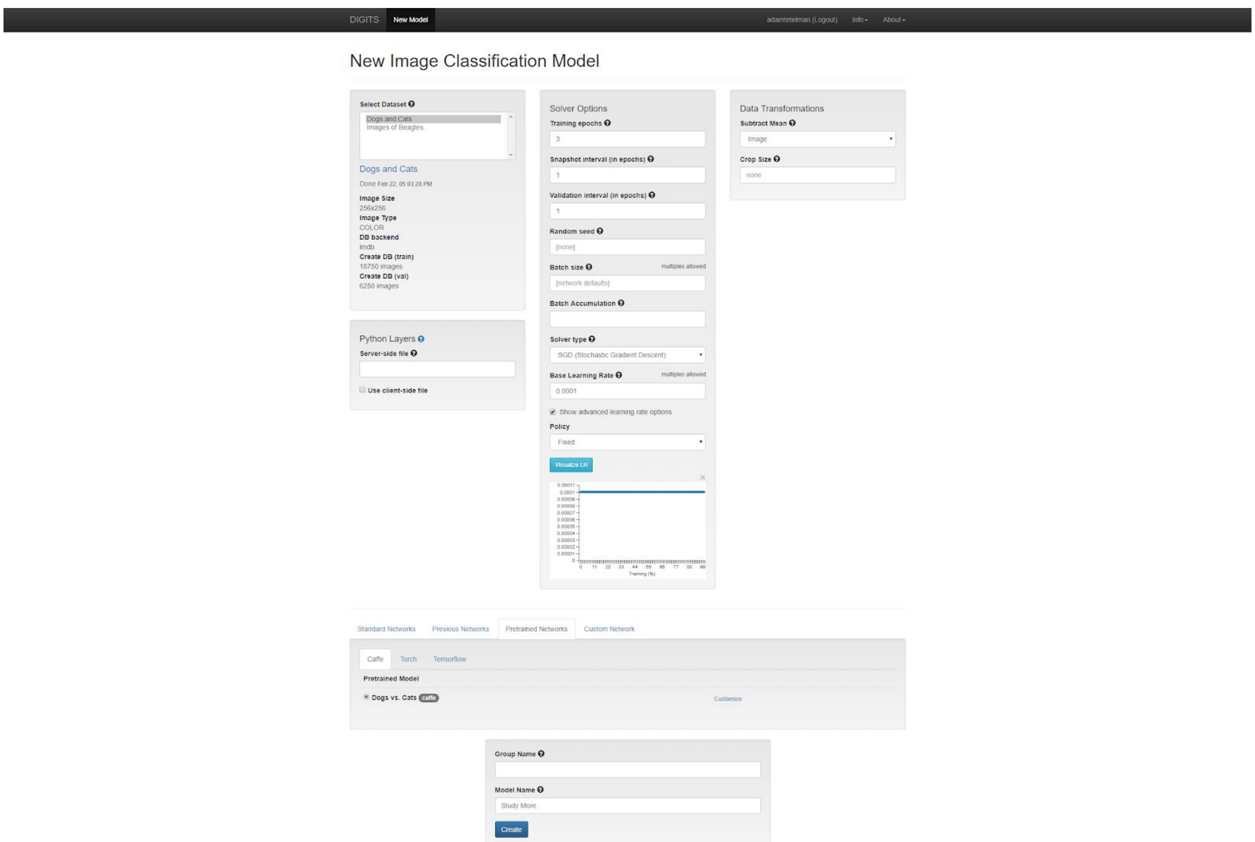
2) **Hyperparameters** - Making changes to options like learning rate are like changing your training "style." Currently, finding the right hyperparameters is a manual process learned through experimentation. As you build intuition about what types of jobs respond well to what hyperparameters, your performance 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 can not be the only intervention you apply.

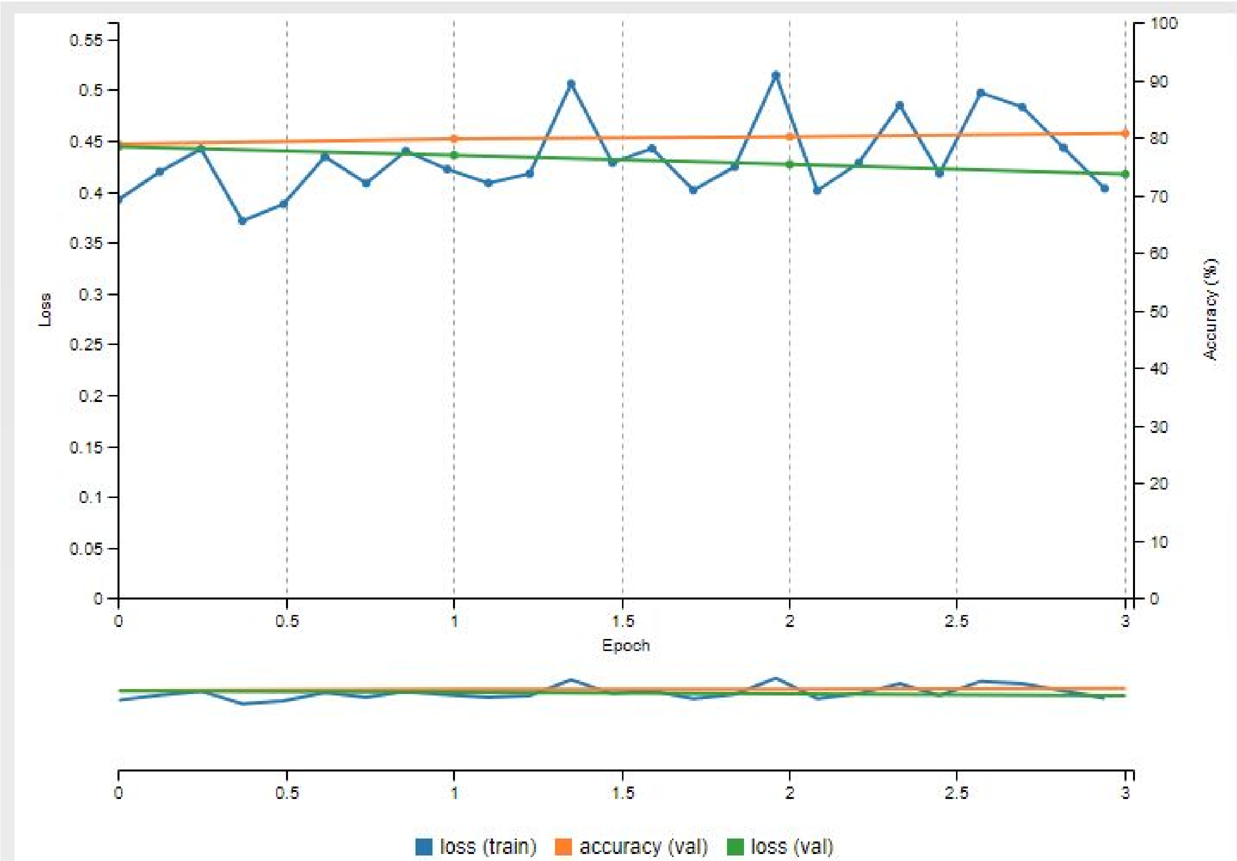
4) **Network architecture** - We'll begin to experiment with network architecture in the next section. This is listed as the last intervention to push back against a false myth that to engage in solving problems with deep learning, people need mastery of network architecture. This field is fascinating and powerful, and improving your skills is a study in math.

Task 4: Performance during Training

We saved our previous model as pretrained model. Based on our pretrained model, we changed the learning rate to 0.0001 and instead of choosing a "Standard Network," select "Pretrained Networks" in order to build the new model on it. And training with the same dataset -- “Dogs vs. Cats”.



Here is the result below:



1. As expected, the accuracy starts close to where our first model left off, 80%.
2. Accuracy DOES continue to increase, showing that increasing the number of epochs often does increase performance.
3. 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.

After working on our retrained model, it is more reasonable to check out other models that created by experts from “ImageNet”. 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.
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
```

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
```

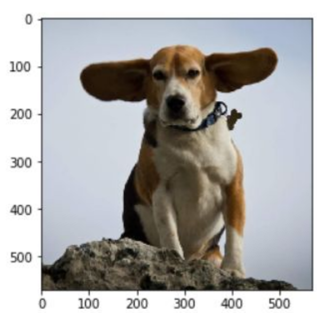
By deploying the model from “ImageNet”, we got an array with the probability for 1000 classes, and, finialy, we refined the result to classify the picture as “beagle” dog.

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

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

output
```

```
{'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,
          8.94371688e-10,   2.96192459e-10,   3.58631702e-09,
```

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

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

```
Input image:
```

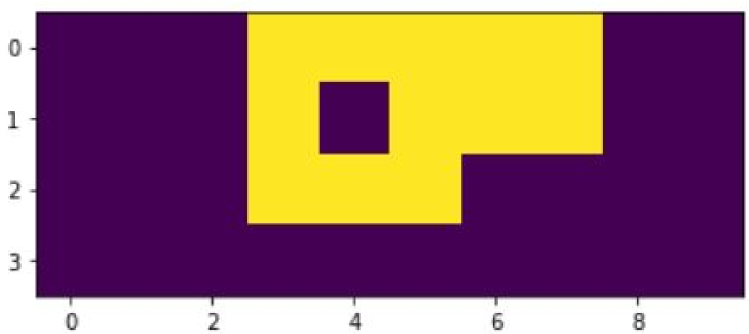



```
Output label:n02088364 beagle
```

Task 5: Object Detection

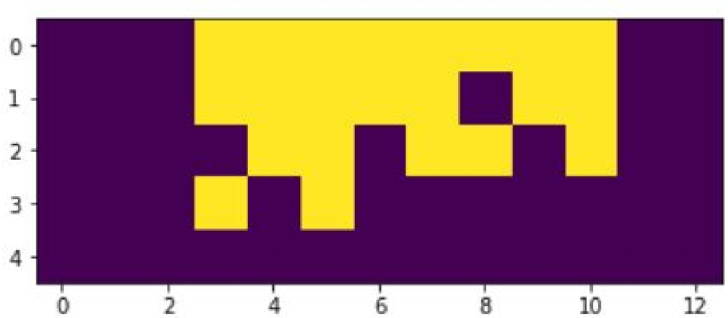
Concept: This task is aimed to solving an object detection problem will be to combine an image classification network with traditional programming to create the input/output pairing; using the "sliding window" approach, where taking split out image into small

sections which called grid squares. If that grid square contains an image of a dog, we'll have localized Louie in the image.
Experiment:
1. Using Deployment
Using the sliding window approach involves deploying an image classifier trained to classify 256X256 portions of an image as either Louie or not.
non-overlapping grid squares:



25% overlapping grid squares:

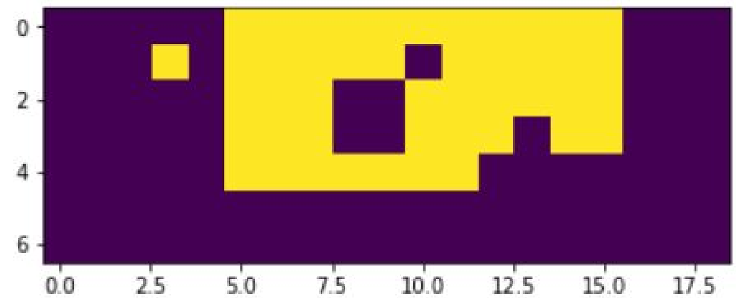
```
Total inference time (sliding window without overlap): 2.8700299263 seconds
Image has 4*10 blocks of 256 pixels
With overlap=0.250000 grid_size=5*13
```




```
Total inference time (sliding window with 25.000000% overlap: 4.49907684326 seconds
```

50% overlapping grid squares:

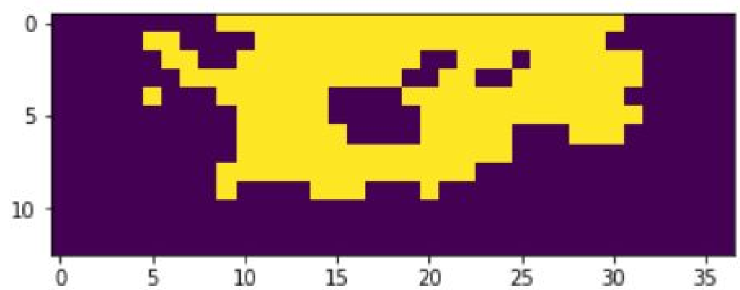
```
Image has 4*10 blocks of 256 pixels
With overlap=0.500000 grid_size=7*19
```




```
Total inference time (sliding window with 50.000000% overlap: 9.13731598854 seconds
```

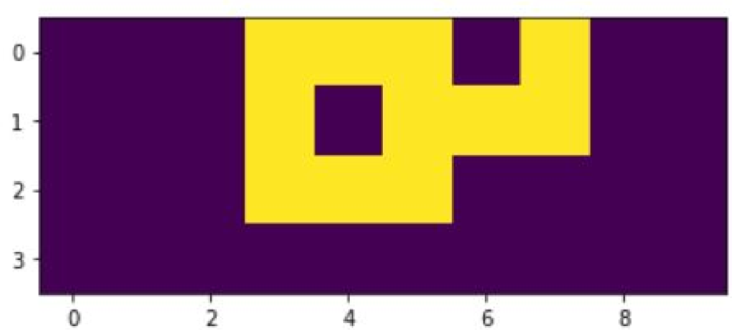
75% overlapping grid squares:

```
Total inference time (sliding window without overlap): 2.9044919014 seconds
Image has 4*10 blocks of 256 pixels
With overlap=0.750000 grid_size=13*37
```




```
Total inference time (sliding window with 75.000000% overlap: 34.0976500511 seconds
```

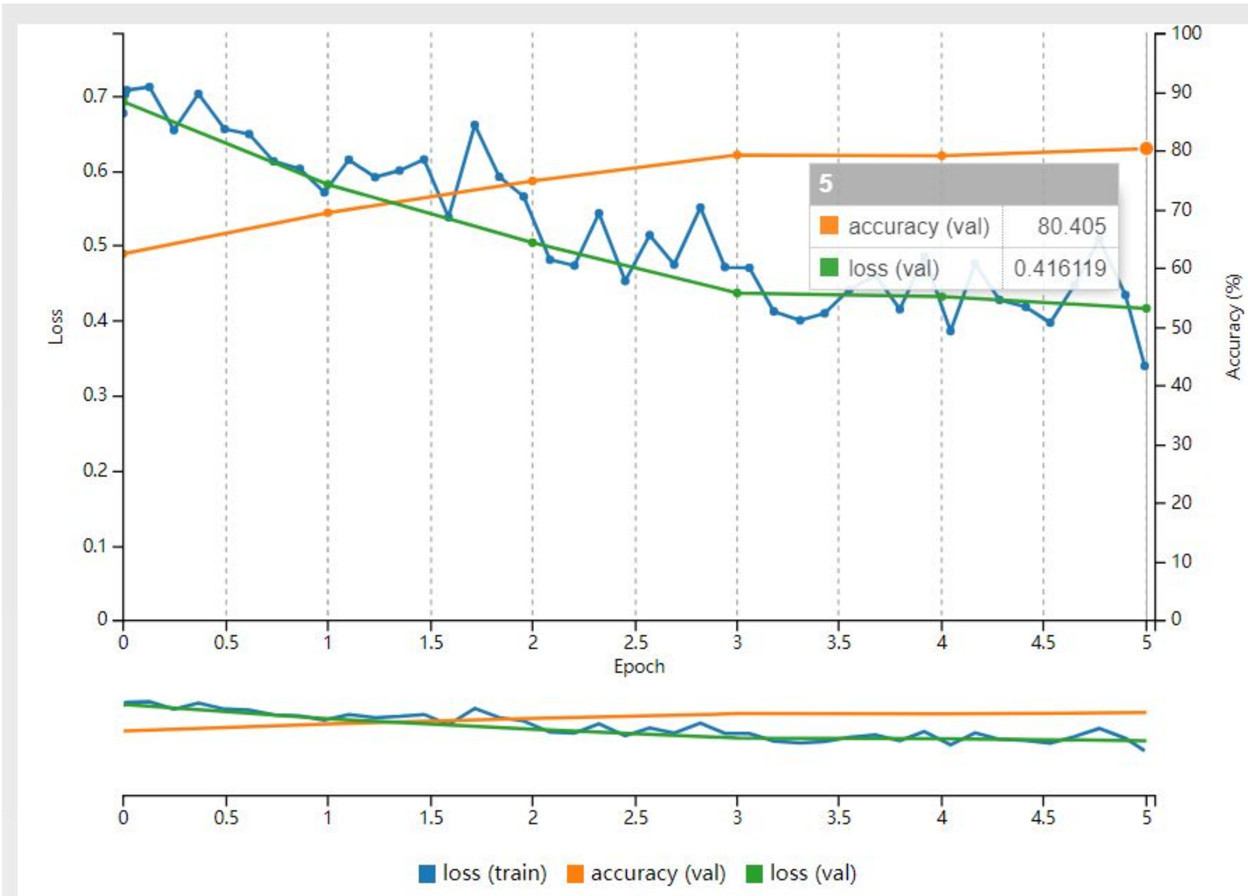
Banched interface:



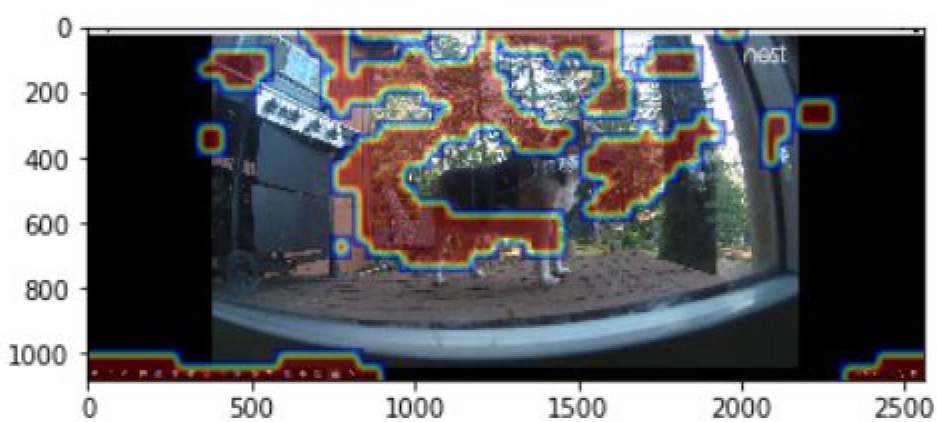

```
Total inference time (batched inference): 0.143352985382 seconds
```

We can see that the higher the overlapping rate, the more detail detection it can get.

2. Rebuilding from an existing neural network
Converting AlexNet into a Fully-Convolutional Network



Using Fully-Convolutional Network to detect the dog. The input shape can be randomly selected.

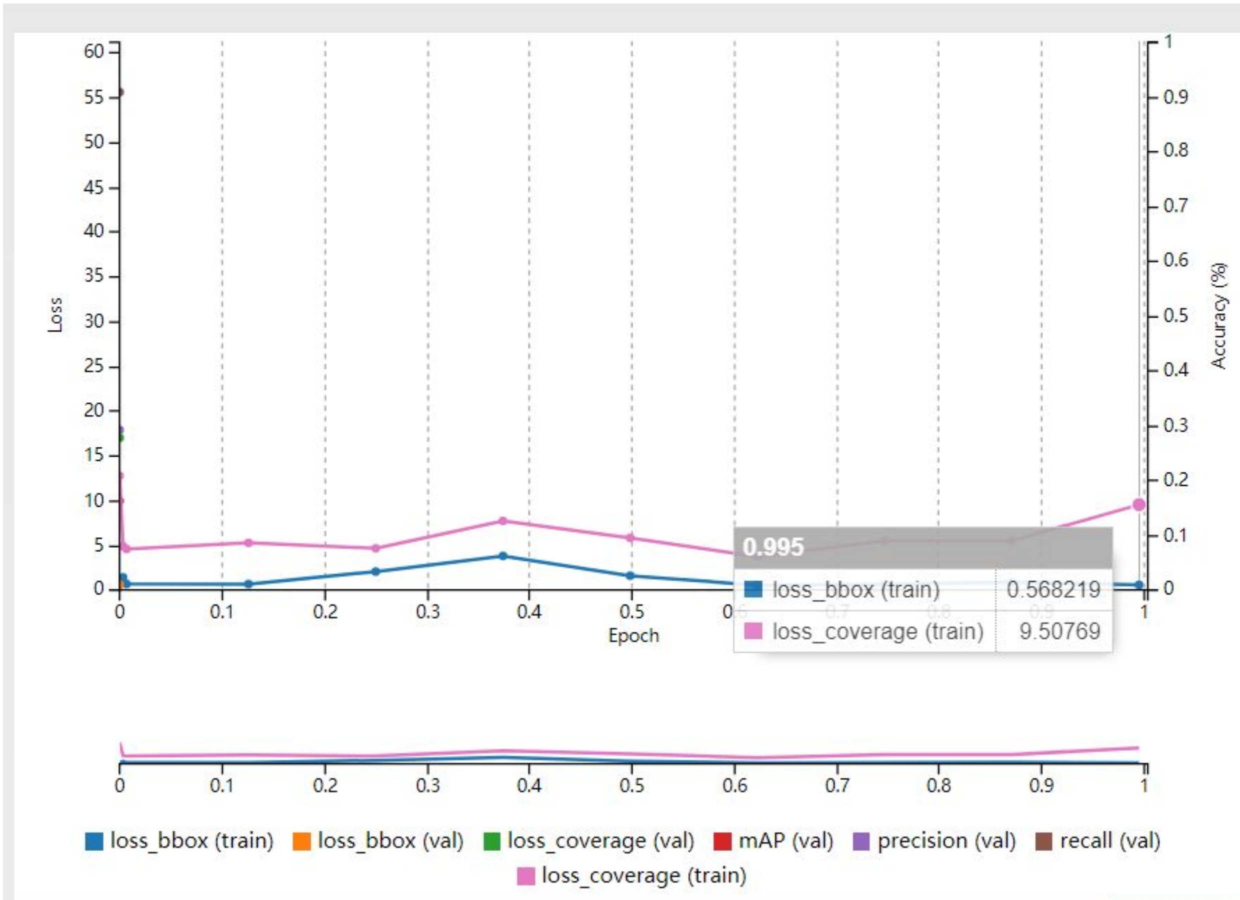



```
Total inference time: 0.342292070389 seconds
```

It shows better performance than sliding window.

3. DetectNet

Training a DetectNet with epho = 1 and learning rate = 0.0001



Test an image to see if Louie is detected

Test another image that not Louie

bbox-list

Task 6: Train and Deploy Neural Networks

1. Create whale faces dataset

Job Information

Job Directory
/dli/data/digits/20190216-211010-69b1

Image Dimensions
256x256 (Width x Height)
Image Type
Color
Resize Transformation
Squash
DB Backend
lmdb
Image Encoding
png
DB Compression
none

Dataset size
645 MB

Parse Folder (train/val)

Folder
/dli/data/whale/data/train

Number of Categories
2
Training Images
6814
Validation Images
2272 (25.0%)

2. Set the model parameters

Solver Options

Training epochs
30

Snapshot interval (in epochs)
1.0

Validation interval (in epochs)
1.0

Random seed
[none]

Batch size multiples allowed
[network defaults]

Batch Accumulation

Blob format
NVCaffe

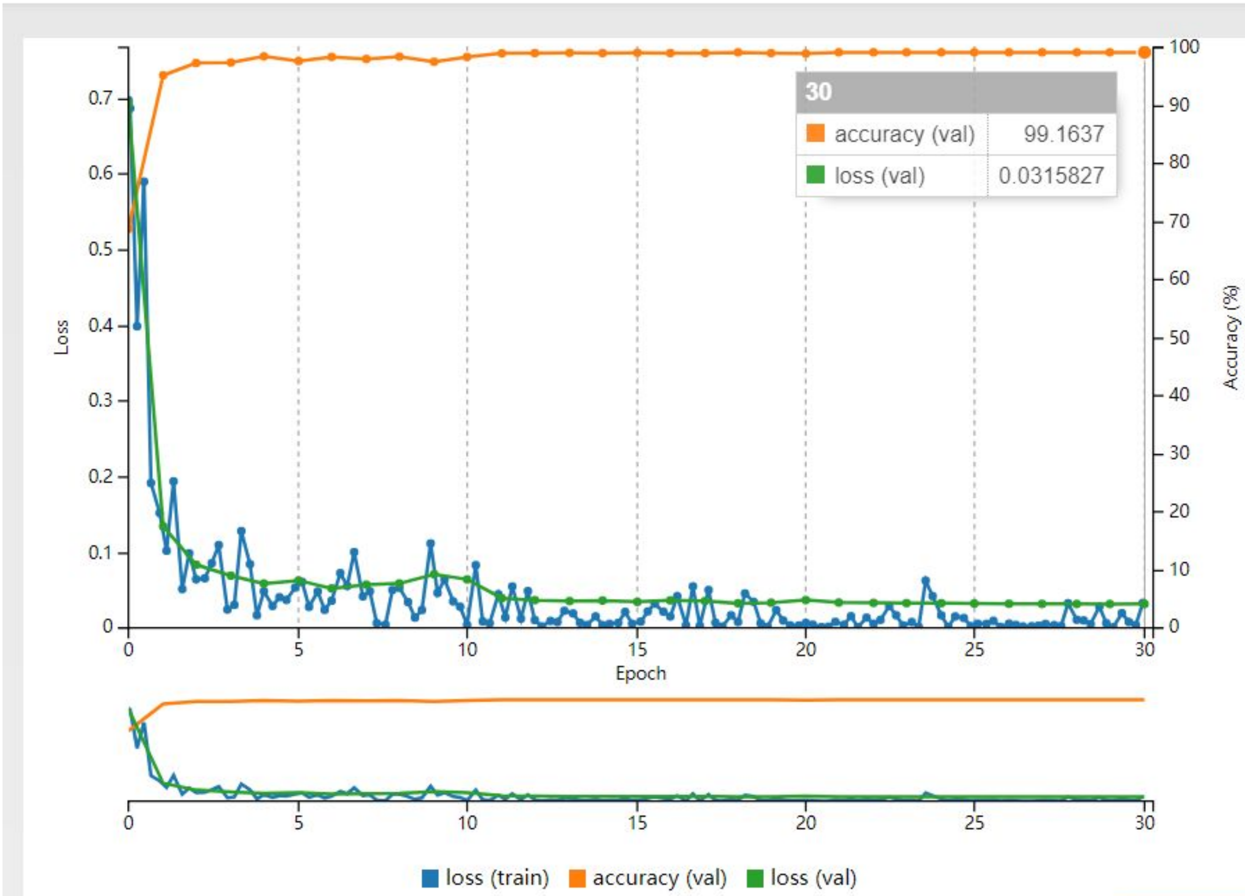
Solver type
SGD (Stochastic Gradient Descent)

Base Learning Rate multiples allowed
0.01

☐ Show advanced learning rate options

In this task I selected AlexNet as training network.

3. Assess the model



The accuracy of validation set is more than 99%. So the model works perfectly without overfitting.

4. Test single image

```
In [2]: !python submission.py '/dli/data/whale/data/train/face/w_1.jpg'  #This should return "whale" at the very bottom
```

```
I0216 21:35:05.731096   191 net.cpp:1137] Copying source layer conv2 Type:Convolution #blobs=2
I0216 21:35:05.731284   191 net.cpp:1137] Copying source layer relu2 Type:ReLU #blobs=0
I0216 21:35:05.731298   191 net.cpp:1137] Copying source layer norm2 Type:LRN #blobs=0
I0216 21:35:05.731308   191 net.cpp:1137] Copying source layer pool2 Type:Pooling #blobs=0
I0216 21:35:05.731321   191 net.cpp:1137] Copying source layer conv3 Type:Convolution #blobs=2
I0216 21:35:05.731753   191 net.cpp:1137] Copying source layer relu3 Type:ReLU #blobs=0
I0216 21:35:05.731768   191 net.cpp:1137] Copying source layer conv4 Type:Convolution #blobs=2
I0216 21:35:05.732105   191 net.cpp:1137] Copying source layer relu4 Type:ReLU #blobs=0
I0216 21:35:05.732120   191 net.cpp:1137] Copying source layer conv5 Type:Convolution #blobs=2
I0216 21:35:05.732357   191 net.cpp:1137] Copying source layer relu5 Type:ReLU #blobs=0
I0216 21:35:05.732372   191 net.cpp:1137] Copying source layer pool5 Type:Pooling #blobs=0
I0216 21:35:05.732380   191 net.cpp:1137] Copying source layer fc6 Type:InnerProduct #blobs=2
I0216 21:35:05.749689   191 net.cpp:1137] Copying source layer relu6 Type:ReLU #blobs=0
I0216 21:35:05.749723   191 net.cpp:1137] Copying source layer drop6 Type:Dropout #blobs=0
I0216 21:35:05.749733   191 net.cpp:1137] Copying source layer fc7 Type:InnerProduct #blobs=2
I0216 21:35:05.757314   191 net.cpp:1137] Copying source layer relu7 Type:ReLU #blobs=0
I0216 21:35:05.757344   191 net.cpp:1137] Copying source layer drop7 Type:Dropout #blobs=0
I0216 21:35:05.757349   191 net.cpp:1137] Copying source layer fc8 Type:InnerProduct #blobs=2
I0216 21:35:05.757372   191 net.cpp:1129] Ignoring source layer loss
whale
```

```
In [3]: !python submission.py '/dli/data/whale/data/train/not_face/w_1.jpg'  #This should return "not whale" at the very bottom
```

```
I0216 21:35:12.759181   206 net.cpp:1137] Copying source layer conv2 Type:Convolution #blobs=2
I0216 21:35:12.759373   206 net.cpp:1137] Copying source layer relu2 Type:ReLU #blobs=0
I0216 21:35:12.759388   206 net.cpp:1137] Copying source layer norm2 Type:LRN #blobs=0
I0216 21:35:12.759398   206 net.cpp:1137] Copying source layer pool2 Type:Pooling #blobs=0
I0216 21:35:12.759413   206 net.cpp:1137] Copying source layer conv3 Type:Convolution #blobs=2
I0216 21:35:12.759884   206 net.cpp:1137] Copying source layer relu3 Type:ReLU #blobs=0
I0216 21:35:12.759902   206 net.cpp:1137] Copying source layer conv4 Type:Convolution #blobs=2
I0216 21:35:12.760273   206 net.cpp:1137] Copying source layer relu4 Type:ReLU #blobs=0
I0216 21:35:12.760289   206 net.cpp:1137] Copying source layer conv5 Type:Convolution #blobs=2
I0216 21:35:12.760529   206 net.cpp:1137] Copying source layer relu5 Type:ReLU #blobs=0
I0216 21:35:12.760545   206 net.cpp:1137] Copying source layer pool5 Type:Pooling #blobs=0
I0216 21:35:12.760555   206 net.cpp:1137] Copying source layer fc6 Type:InnerProduct #blobs=2
I0216 21:35:12.777652   206 net.cpp:1137] Copying source layer relu6 Type:ReLU #blobs=0
I0216 21:35:12.777686   206 net.cpp:1137] Copying source layer drop6 Type:Dropout #blobs=0
I0216 21:35:12.777694   206 net.cpp:1137] Copying source layer fc7 Type:InnerProduct #blobs=2
I0216 21:35:12.785291   206 net.cpp:1137] Copying source layer relu7 Type:ReLU #blobs=0
I0216 21:35:12.785320   206 net.cpp:1137] Copying source layer drop7 Type:Dropout #blobs=0
I0216 21:35:12.785328   206 net.cpp:1137] Copying source layer fc8 Type:InnerProduct #blobs=2
I0216 21:35:12.785364   206 net.cpp:1129] Ignoring source layer loss
not whale
```

I also tested another two images, all of them are correctly classified.

Predictions

face	100.0%
not face	0.0%

Predictions

not face	99.96%
face	0.04%