Homework #2

Task:	Use Python + a caffe-trained CNN to classify images.
Due Date:	5 p.m. EST, Friday, October 27, 2017
Programming Language:	Python + Caffe
Dataset:	ImageNet subset (4 classes)

Guide to dataset

ImageNet has ~14M images and 1000 classes. For this assignment, we will be using 1280 images and four classes:

Class Name	Class Label	Identifier	Number of Images
Fox	0	n02119789	320
Bike	1	n03782782	320
Elephant	2	n02504458	320
Car	3	n04037443	320

A few notes:

- Class labels are 0-indexed in caffe
- Images come in all resolutions. (Lucky for you, caffe warps them to the input size for free.)
- To get a feel for the dataset, navigate to the image folders and look at some of the images.

Instructions:

A) Create a hw2-train.prototxt and hw2-solver.prototxt file.
B) Train your network. Create the hw2-weights.caffemodel file.
C) Create a hw2-deploy.prototxt and hw2.py file and test your network.
D) Submit files to Blackboard for grading.

Grading Criteria:

50 points	Your hw2-deploy.prototxt and hw2-weights.caffemodel load without error
30 points	Your model performs a forward pass and provides class predictions
20 points	Your model predicts the correct proposal 50% of the time (chance is 25%).

Intro to CNNs

A standard CNN takes a 2D image as input and produces a class prediction. In addition to fully-connected and activation layers (that we used in homework 1), we use convolutional and pooling layers. A standard CNN network might look something like Figure 1 (which happens to be the "VGG-16 architecture").

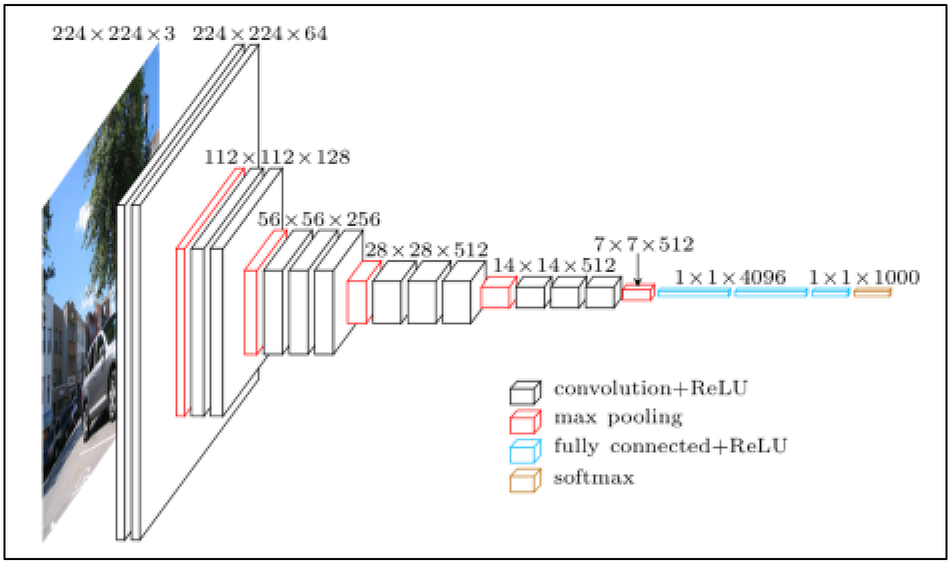


Figure 1 - VGG16 (Source: https://www.cs.toronto.edu/~frossard/post/vgg16/)

Let's explain this diagram in some greater detail:

- We start with an image of resolution 224x224x3 (3 = RGB color channels)
- The first layer is a convolutional layer. It convolves over the entire image and produces an output of the same resolution. It is followed by the ReLU activation function.
  - (In this image, the "convolutional layer" and its activation function (ReLU) are drawn as one layer. Calling them "one layer" is pretty standard.)
- You'll notice that the first two "convolution + ReLU" layers are written as 224x224x64. The "64" represents the number of "filters". That means there are 64 filters of resolution 224x224 in the first layer…and the same number of filters in the second layer. The number of "filters" you have at each layer is a choice you make (just as you chose how many nodes to include in your hidden layer in homework #1).
- After the second convolutional layer, we come to a max pooling layer. The max pooling layer shrinks each dimension by some factor (usually 2). This means that a 2x2 set of pixels become 1 "pixel".
- This process repeats for several "groups" until we get to the last max pooling layer (of dimension 7x7). Then comes a fully connected layer (with 4096 nodes). Each node in the fully connected layer is connected to every "node" from the previous layer (7x7x512).
- The first two blue tubes are of the same dimensions. They are followed by a fully-connected layer with 1000 nodes and a Softmax layer.

If you find you want to understand all of this a little better, a good resource can be found here: http://cs231n.github.io/convolutional-networks/. The sections called "Convolution Demo" and "General Pooling" have some great images to illustrate how each of these layers work.

A final note: CNN's have a fixed input size. Since the images come in all types of resolutions, we warp them to that fixed size. This means that some images might be fed into the network "squished" and others might be "stretched". That's a pretty standard approach to CNN's.

Guide to getting started.

1. Go to a computer lab of your choice (e.g. MSEE189, EE206, EE207).
2. Sit down at a Linux computer and login.
3. Open a terminal.
4. Type “ee570” and press <enter>.

What just happened when I did that?

- We loaded an environment for you…and ran a script.
    - The environment lets you run caffe by setting some environmental variables.
    - The script creates a directory for you to work in, copied over some files, and printed some instructions. Details about the files will be provided in the subsequent sections.

Part A: Create a hw2-train.prototxt and hw2-solver.prototxt file
Caffe is not really a "NN programming language". Instead, it is a tool (written in C++) that people use to train their network. They then use a wrapper (usually python) to "deploy" and test the network.

While training, caffe reports the loss (just like in Torch). This allows you to keep an eye on how training is progressing to ensure it is moving toward a solution.

You can also configure caffe to test (and/or validate) the network while training. What this means is that at some specified interval (e.g. after every 5 epochs), the network will run a bunch of forward passes on your test_split and compute the accuracy. This is something else you can keep your eye on to see how training is progressing…and to see if you are moving towards a solution. (This is also a good way to make sure you aren't overfitting.)

To train a network using caffe, you need to create two files:

hw2-train.prototxt
hw2-solver.prototxt

In the hw2-train.prototxt file, you define the shape of our network. As part of this file, you specify any hyperparameters that are specific to the layer.

- An example of the layers and hyperparameters you will need, along with descriptions of what each one is, can be found in the hw2-guide-train.prototxt file that was copied over.

In the hw2-solver.prototxt file, you describe the "solver" parameters used during training. These include the learning rate, how long to train for, where to save off snapshots of your network, etc.

- An example of the relevant hyperparameters, along with descriptions of what each one is, can be found in hw2-guide-solver.prototxt file that was copied over.

Some rules:

- Your NN should take a 32x32x3 image as input.
- Your NN's first layer must have the name "data".
- Your NN should end with a FC with 4 output nodes followed by a Softmax layer.
- The Softmax layer must have the name "prob".

Some advice:

- Designing your network architecture may be the hardest part of this homework. You need to balance two things: (1) you want it to be large enough that it has the representational capacity to learn the problem and (2) you want it to be small enough that it doesn't take a year to train.
- The homework was designed such that you should be able to train a network on ecegrid that gets above 60% accuracy in <5 minutes. (I was able to reach 78% test accuracy in 4 minutes.)
- Don't try and be a hero and get >90% accuracy. Getting above 90% (or even 80%) may be impossible because of the tiny input size (32x32 pixels) and the small computational power of ecegrid (which forces us to use simpler NN's when there are more robust architectures out there).

Some networking architecture advice:

- You want the network to have the representational capacity to learn the problem. I tried using the lenet5 architecture. It seemed incapable of learning the problem. You'll want to use something bigger than that.

- You want the network to be small enough that it doesn’t take a year to train. I tried using the VGG-16 network. It was able to learn the problem…but took 9 hours to train.
- The network I finally settled on had a data layer, 3 conv layers, 3 fully-connected layers, and a Softmax layer. I’d use something close to that.
- For the conv layers, you don’t need a kernel larger than 3x3 (although you are welcome to experiment with it).
- For the pool layers, you don’t need a kernel larger than 2x2 (although you are welcome to experiment with it). The problem with collapsing your dimensions too quickly is that you may lose information that is useful to solving the problem.
- For the fully-connected layers, you don’t need more than 2k nodes per layer.
- If you want to see what your network looks like, you can paste it into this tool (and press <shift> + <enter> to see it) : http://ethereon.github.io/netscope/#/editor.
    - This will also let you know if you have a “syntax error” somewhere in the file.

Part B: Train your network

Once these two files are finished, you train your network by typing this into the command line:

```
caffe train --solver=<pathToSolverFile>
```

When that command is invoked, caffe will load your hw2-train.prototxt file and spit out a bunch of information to the console (shape of train network, shape of test network, memory calculations, etc.) Unless there is an error, you shouldn't worry about the output.

After the network is loaded, caffe will begin training and you'll see output that looks something like this:

```
I0928 09:56:02.624604  3392 caffe.cpp:248] Starting Optimization
I0928 09:56:02.624610  3392 solver.cpp:272] Solving
I0928 09:56:02.624614  3392 solver.cpp:273] Learning Rate Policy: fixed
I0928 09:56:02.684284  3392 solver.cpp:330] Iteration 0, Testing net (#0)
I0928 09:56:02.684973  3392 blocking_queue.cpp:49] Waiting for data
I0928 09:56:03.883700  3392 solver.cpp:397]     Test net output #0: accuracy/top-1 = 0.3
I0928 09:56:03.883735  3392 solver.cpp:397]     Test net output #1: loss = 25.5634 (* 1 = 25.5634 loss)
I0928 09:56:04.453475  3392 solver.cpp:218] Iteration 0 (-1.11303e-38 iter/s, 1.828s/10 iters), loss = 22.5997
I0928 09:56:04.453521  3392 solver.cpp:237]     Train net output #0: loss = 22.5997 (* 1 = 22.5997 loss)
I0928 09:56:04.453532  3392 sgd_solver.cpp:105] Iteration 0, lr = 0.0005
I0928 09:56:10.421917  3392 solver.cpp:218] Iteration 10 (1.6756 iter/s, 5.968s/10 iters), loss = 4.40518
I0928 09:56:10.421970  3392 solver.cpp:237]     Train net output #0: loss = 4.40518 (* 1 = 4.40518 loss)
I0928 09:56:10.421986  3392 sgd_solver.cpp:105] Iteration 10, lr = 0.0005
I0928 09:56:17.959875  3392 solver.cpp:218] Iteration 20 (1.32679 iter/s, 7.537s/10 iters), loss = 1.41027
I0928 09:56:17.959929  3392 solver.cpp:237]     Train net output #0: loss = 1.41027 (* 1 = 1.41027 loss)
I0928 09:56:17.959951  3392 sgd_solver.cpp:105] Iteration 20, lr = 0.0005
I0928 09:56:25.325260  3392 solver.cpp:218] Iteration 30 (1.35777 iter/s, 7.365s/10 iters), loss = 1.21616
I0928 09:56:25.325314  3392 solver.cpp:237]     Train net output #0: loss = 1.21616 (* 1 = 1.21616 loss)
I0928 09:56:25.325331  3392 sgd_solver.cpp:105] Iteration 30, lr = 0.0005
I0928 09:56:34.089659  3392 solver.cpp:218] Iteration 40 (1.14103 iter/s, 8.764s/10 iters), loss = 0.993588
I0928 09:56:34.089802  3392 solver.cpp:237]     Train net output #0: loss = 0.993589 (* 1 = 0.993589 loss)
I0928 09:56:34.089818  3392 sgd_solver.cpp:105] Iteration 40, lr = 0.0005
I0928 09:56:38.896281  3392 solver.cpp:218] Iteration 50 (2.08073 iter/s, 4.806s/10 iters), loss = 1.08927
I0928 09:56:38.896332  3392 solver.cpp:237]     Train net output #0: loss = 1.08927 (* 1 = 1.08927 loss)
I0928 09:56:38.896347  3392 sgd_solver.cpp:105] Iteration 50, lr = 0.0005
I0928 09:56:43.761076  3392 solver.cpp:218] Iteration 60 (2.05592 iter/s, 4.864s/10 iters), loss = 0.902794
I0928 09:56:43.761129  3392 solver.cpp:237]     Train net output #0: loss = 0.902795 (* 1 = 0.902795 loss)
I0928 09:56:43.761147  3392 sgd_solver.cpp:105] Iteration 60, lr = 0.0005
I0928 09:56:48.719136  3392 solver.cpp:218] Iteration 70 (2.01694 iter/s, 4.958s/10 iters), loss = 0.961684
I0928 09:56:48.719190  3392 solver.cpp:237]     Train net output #0: loss = 0.961685 (* 1 = 0.961685 loss)
I0928 09:56:48.719208  3392 sgd_solver.cpp:105] Iteration 70, lr = 0.0005
I0928 09:56:51.696126  3392 solver.cpp:330] Iteration 77, Testing net (#0)
I0928 09:56:52.692442  3392 solver.cpp:397]     Test net output #0: accuracy/top-1 = 0.483333
I0928 09:56:52.692487  3392 solver.cpp:397]     Test net output #1: loss = 1.12028 (* 1 = 1.12028 loss)
I0928 09:56:54.707960  3392 solver.cpp:218] Iteration 80 (1.67001 iter/s, 5.988s/10 iters), loss = 0.787796
I0928 09:56:54.708012  3392 solver.cpp:237]     Train net output #0: loss = 0.787797 (* 1 = 0.787797 loss)
I0928 09:56:54.708030  3392 sgd_solver.cpp:105] Iteration 80, lr = 0.0005
I0928 09:56:59.597609  3392 solver.cpp:218] Iteration 90 (2.04541 iter/s, 4.889s/10 iters), loss = 0.698227
I0928 09:56:59.597661  3392 solver.cpp:237]     Train net output #0: loss = 0.698228 (* 1 = 0.698228 loss)
I0928 09:56:59.597677  3392 sgd_solver.cpp:105] Iteration 90, lr = 0.0005
I0928 09:57:04.531232  3392 solver.cpp:218] Iteration 100 (2.02716 iter/s, 4.933s/10 iters), loss = 0.654892
I0928 09:57:04.531338  3392 solver.cpp:237]     Train net output #0: loss = 0.654893 (* 1 = 0.654893 loss)
I0928 09:57:04.531355  3392 sgd_solver.cpp:105] Iteration 100, lr = 0.0005
I0928 09:57:09.452517  3392 solver.cpp:218] Iteration 110 (2.03211 iter/s, 4.921s/10 iters), loss = 1.13311
I0928 09:57:09.452571  3392 solver.cpp:237]     Train net output #0: loss = 1.13311 (* 1 = 1.13311 loss)
I0928 09:57:09.452587  3392 sgd_solver.cpp:105] Iteration 110, lr = 0.0005
I0928 09:57:14.314335  3392 solver.cpp:218] Iteration 120 (2.05719 iter/s, 4.861s/10 iters), loss = 0.835117
I0928 09:57:14.314388  3392 solver.cpp:237]     Train net output #0: loss = 0.835117 (* 1 = 0.835117 loss)
I0928 09:57:14.314404  3392 sgd_solver.cpp:105] Iteration 120, lr = 0.0005
I0928 09:57:19.266557  3392 solver.cpp:218] Iteration 130 (2.01939 iter/s, 4.952s/10 iters), loss = 0.64101
I0928 09:57:19.266610  3392 solver.cpp:237]     Train net output #0: loss = 0.641011 (* 1 = 0.641011 loss)
I0928 09:57:19.266626  3392 sgd_solver.cpp:105] Iteration 130, lr = 0.0005
I0928 09:57:24.175053  3392 solver.cpp:218] Iteration 140 (2.03749 iter/s, 4.908s/10 iters), loss = 0.570962
I0928 09:57:24.175106  3392 solver.cpp:237]     Train net output #0: loss = 0.570963 (* 1 = 0.570963 loss)
I0928 09:57:24.175122  3392 sgd_solver.cpp:105] Iteration 140, lr = 0.0005
I0928 09:57:29.077868  3392 solver.cpp:218] Iteration 150 (2.03998 iter/s, 4.902s/10 iters), loss = 0.573994
```

```
I0928 09:57:29.077922  3392 solver.cpp:237]     Train net output #0: loss = 0.573995 (* 1 = 0.573995 loss)
I0928 09:57:29.077944  3392 sgd_solver.cpp:105] Iteration 150, lr = 0.0005
I0928 09:57:30.592715  3392 solver.cpp:330] Iteration 154, Testing net (#0)
I0928 09:57:31.590024  3392 solver.cpp:397]     Test net output #0: accuracy/top-1 = 0.675
I0928 09:57:31.590070  3392 solver.cpp:397]     Test net output #1: loss = 0.847037 (* 1 = 0.847037 loss)
```

As caffe trains, it will save off "snapshots" of the weight values. It saves off snapshots at the frequency you specify in the hw2-solver.prototxt file. The filenames have this format:

    <snapshotPrefix>_iter_<someNumber>.caffemodel

You can save off weight files as often as you want. When it comes time for testing, it's best to choose the one that has the best accuracy.

- How do you know which weight file has the best accuracy? Watch the console and the test accuracy over time. Once training completes, find the "iteration" at which the test accuracy was highest. Then find the snapshot that was saved as close to that iteration as possible. This weights file that should have the best accuracy.
- NOTE: In your hw2-solver.prototxt file, you can configure caffe to save off the weights file at the same frequency that you test. That makes it easy to choose which weights file has the best accuracy.
- NOTE: The weights files can be decently large. (For example, the weight file from the VGG-16 network can be 500 MB…so you don't want to save *too* often or you'll fill up your hard drive with snapshots.)

You can resume training from a snapshot. To do so, you add the "--snapshot" command line argument with a path to the <absPathTo_iter###>.solverstate file. For example:

```
caffe train --solver=/home/jjohanse/ece570/hw2/hw2-solver.prototxt --
snapshot=/home/jjohanse/ece570/hw2/hw2_iter_300.solverstate
```

The <fileName>.caffemodel saves the values of the weights.
The <filename>.solverstate saves the state of the training (e.g. the learning rate).

When training is complete, save the weights file as "hw2-weights.caffemodel". (This is one of the files you will submit as part of your homework.)

Some advice:

- Watch the loss. If it increases (to something like 87) and sticks there, the network is "stuck". It's never going to decrease. Start over. If this repeats, your learning rate may be too high.
- Watch the loss. If it doesn't decrease after a few minutes, it may never decrease sufficiently. On explanation is that your network architecture may be too simple. (This is what happened when I experimented with the lenet5 architecture.)
- Watch the loss. It sometimes bounces around (up and down). This is normal. Look at the loss in the console output I included above. You can see it bouncing around (quite a bit) as it moved towards a solution.
- Watch the loss and the accuracy. This is the best way to see if the network is doing what it is supposed to.

Part C: Create a hw2-deploy.prototxt and hw2.py file and test your network
Caffe does not provide you with the ability to supply a single image and get a single predicted output. To do this, you have to use a wrapper (Python, Matlab, or C++). We will be using Python.

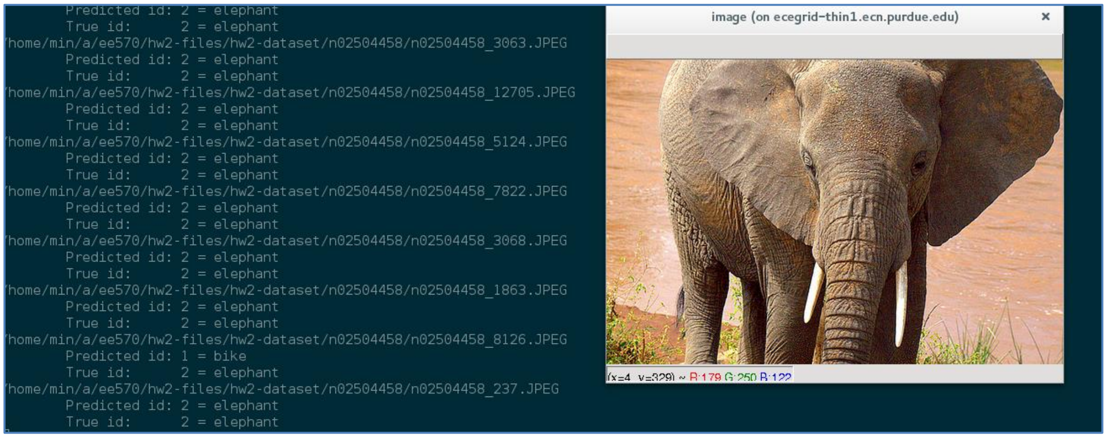
Here's the general idea:

(A) Load your CNN.
(B) Loop over the images in your test split.
(C) Feed an individual image into the network and perform a forward pass.
(D) Take the outputs and determine which class was predicted.
(E) Compute the accuracy over the entire test split. (It should be nearly the same as what caffe reported on the command line.)

A few notes:

(a) In order to load your CNN in python, you need to create a hw2-deploy.prototxt file. This file is nearly identical to your hw2-train.prototxt file. You should copy and paste it into the new file. Then you make two tweaks:
    (1) Replace the "Image" layers with an input layer.
    (2) Get rid of the "loss" and "accuracy" layers at the end of the network (and any layers that may rely on a "label" blob).
An example of this is provided in the hw2-guide-deploy.prototxt file.
(b) Make sure the names of your layers in your hw2-deploy.prototxt match the names of the layers in your hw2-train.prototxt file. This is necessary for the weights to load properly.
(c) When we were training the network in part B, caffe warped all input images to the proper dimensions for us. Now, we don't have that luxury. We need to warp the image ourselves before inputting it into the network.
    a. An example of how to do this can be found in hw2-guide.py.
(d) Caffe refers to the inputs to each layer as "bottom blobs" and the outputs of each layer as "top blobs". Thus, when we want to input our image into caffe (and have it run a forward pass), we need to put it into the "bottom blob" of the input layer.
    a. An example of how to do this can be found in hw2-guide.py.
(e) When we want to get the predicted classes, we extract them from the "top blob" of the last layer.
    a. An example of how to do this can be found in hw2-guide.py.

Via this python wrapper, you should be able to see your network in action, predicting what class each image is from. Cool! Congrats! You're amazing.



```
       Predicted id: 2 = elephant
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_3063.JPEG
       Predicted id: 2 = elephant
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_12705.JPEG
       Predicted id: 2 = elephant
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_5124.JPEG
       Predicted id: 2 = elephant
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_7822.JPEG
       Predicted id: 2 = elephant
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_3068.JPEG
       Predicted id: 2 = elephant
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_1863.JPEG
       Predicted id: 2 = elephant
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_8126.JPEG
       Predicted id: 1 = bike
       True id:      2 = elephant
/home/min/a/ee570/hw2-files/hw2-dataset/n02504458/n02504458_237.JPEG
       Predicted id: 2 = elephant
       True id:      2 = elephant
```

image (on ecegrid-thin1.ecn.purdue.edu)
×
(x=4, y=329) ~ R:179 G:250 B:122

Part D: Submit files to Blackboard for grading

You need to submit the following files and they need to be named exactly this way:

- hw2-train.prototxt
- hw2-solver.prototxt
- hw2-deploy.prototxt
- hw2-weights.caffemodel
- hw2.py

Put these five files into a folder named exactly this way:

- hw2

<u>Zip</u> this file. (No tarballs, please.)

Upload the zipped file to Blackboard.

The grading script will do the following:

- Unzip the hw2.zip folder and extract the five files.
    - If your files/folders are named wrong, you get 0 points on the homework.
- Load your network (i.e. "hw2-deploy.prototxt")
    - If successful, you earn the points
- Load your weights (i.e. "hw2-weights.caffemodel")
    - If successful, you earn the points
- Loop over 50 test images.
    - Feed image into CNN
    - Perform a forward pass and get a prediction
        - If successful, you earn the points
- Calculate the accuracy
    - If the CNN predicted the correct label 50% of the time, you earn the points (chance = 25%)

ImageNet usage

You are using images from ImageNet as part of this homework. As such, you agree to abide by the following terms and conditions:

> **You** (the "Researcher") has requested permission to use **the ImageNet database** (the "Database") at **Princeton University and Stanford University**. In exchange for such permission, Researcher hereby agrees to the following terms and conditions:
>
> 1. Researcher shall use the Database only for non-commercial research and educational purposes.
> 2. Princeton University and Stanford University make no representations or warranties regarding the Database, including but not limited to warranties of non-infringement or fitness for a particular purpose.
> 3. Researcher accepts full responsibility for his or her use of the Database and shall defend and indemnify the ImageNet team, Princeton University, and Stanford University, including their employees, Trustees, officers and agents, against any and all claims arising from Researcher's use of the Database, including but not limited to Researcher's use of any copies of copyrighted images that he or she may create from the Database.
> 4. Researcher may provide research associates and colleagues with access to the Database provided that they first agree to be bound by these terms and conditions.
> 5. Princeton University and Stanford University reserve the right to terminate Researcher's access to the Database at any time.
> 6. If Researcher is employed by a for-profit, commercial entity, Researcher's employer shall also be bound by these terms and conditions, and Researcher hereby represents that he or she is fully authorized to enter into this agreement on behalf of such employer.
> 7. The law of the State of New Jersey shall apply to all disputes under this agreement.