TinyImageNet and Ensembles

So far, we have only worked with the CIFAR-10 dataset. In this exercise we will introduce the TinyImageNet dataset. You will combine several pretrained models into an ensemble, and show that the ensemble performs better than any individual model.

In [1]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt
from time import time

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Introducing TinyImageNet

The TinyImageNet dataset is a subset of the ILSVRC-2012 classification dataset. It consists of 200 object classes, and for each object class it provides 500 training images, 50 validation images, and 50 test images. All images have been downsampled to 64x64 pixels. We have provided the labels for all training and validation images, but have withheld the labels for the test images.

We have further split the full TinyImageNet dataset into two equal pieces, each with 100 object classes. We refer to these datasets as TinyImageNet-100-A and TinyImageNet-100-B.

To download the data, go into the `cs231n/datasets` directory and run the script `get_tiny_imagenet_splits.sh`. Then run the following code to load the TinyImageNet-100-A dataset into memory.

NOTE: The full TinyImageNet dataset will take up about 490MB of disk space, and loading the full TinyImageNet-100-A dataset into memory will use about 2.8GB of memory.

In [2]:
```
from cs231n.data_utils import load_tiny_imagenet

tiny_imagenet_a = 'cs231n/datasets/tiny-imagenet-100-A'

class_names, X_train, y_train, X_val, y_val, X_test, y_test = load_tiny_imagenet(tiny_imagenet_a)

# Zero-mean the data
mean_img = np.mean(X_train, axis=0)
X_train -= mean_img
X_val -= mean_img
X_test -= mean_img
```

```
loading training data for synset 20 / 100
loading training data for synset 40 / 100
loading training data for synset 60 / 100
loading training data for synset 80 / 100
loading training data for synset 100 / 100
```

TinyImageNet-100-A classes

Since ImageNet is based on the WordNet ontology, each class in ImageNet (and TinyImageNet) actually has several different names. For example "pop bottle" and "soda bottle" are both valid names for the same class. Run the following to see a list of all classes in TinyImageNet-100-A:

In [3]:
```
for names in class_names:
    print ' '.join('"%s"' % name for name in names)
```

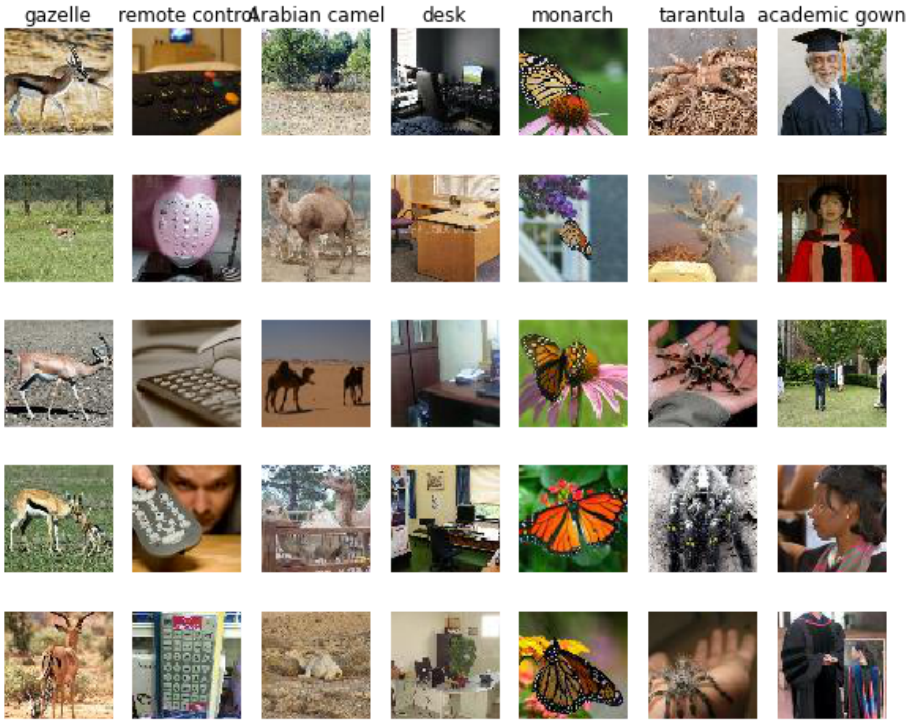
```
"Egyptian cat"
"reel"
"volleyball"
"rocking chair" "rocker"
"lemon"
"bullfrog" "Rana catesbeiana"
"basketball"
"cliff" "drop" "drop-off"
"espresso"
"plunger" "plumber's helper"
"parking meter"
"German shepherd" "German shepherd dog" "German police dog" "alsatian"
"dining table" "board"
"monarch" "monarch butterfly" "milkweed butterfly" "Danaus plexippus"
"brown bear" "bruin" "Ursus arctos"
"school bus"
"pizza" "pizza pie"
"guinea pig" "Cavia cobaya"
"umbrella"
"organ" "pipe organ"
"oboe" "hautboy" "hautbois"
"maypole"
"goldfish" "Carassius auratus"
"potpie"
"hourglass"
"seashore" "coast" "seacoast" "sea-coast"
"computer keyboard" "keypad"
"Arabian camel" "dromedary" "Camelus dromedarius"
"ice cream" "icecream"
"nail"
"space heater"
"cardigan"
"baboon"
"snail"
"coral reef"
"albatross" "mollymawk"
"spider web" "spider's web"
"sea cucumber" "holothurian"
"backpack" "back pack" "knapsack" "packsack" "rucksack" "haversack"
"Labrador retriever"
"pretzel"
"king penguin" "Aptenodytes patagonica"
"sulphur butterfly" "sulfur butterfly"
"tarantula"
"lesser panda" "red panda" "panda" "bear cat" "cat bear" "Ailurus fulgens"
"pop bottle" "soda bottle"
"banana"
"sock"
"cockroach" "roach"
"projectile" "missile"
"beer bottle"
"mantis" "mantid"
"freight car"
"guacamole"
"remote control" "remote"
"European fire salamander" "Salamandra salamandra"
"lakeside" "lakeshore"
"chimpanzee" "chimp" "Pan troglodytes"
"pay-phone" "pay-station"
"fur coat"
"alp"
"lampshade" "lamp shade"
"torch"
"abacus"
"moving van"
"barrel" "cask"
"tabby" "tabby cat"
"goose"
"koala" "koala bear" "kangaroo bear" "native bear" "Phascolarctos cinereus"
"bullet train" "bullet"
"CD player"
"teapot"
"birdhouse"
"gazelle"
"academic gown" "academic robe" "judge's robe"
```

```
"tractor"
"ladybug" "ladybeetle" "lady beetle" "ladybird" "ladybird beetle"
"miniskirt" "mini"
"golden retriever"
"triumphal arch"
"cannon"
"neck brace"
"sombrero"
"gasmask" "respirator" "gas helmet"
"candle" "taper" "wax light"
"desk"
"frying pan" "frypan" "skillet"
"bee"
"dam" "dike" "dyke"
"spiny lobster" "langouste" "rock lobster" "crawfish" "crayfish" "sea crawfish"
"police van" "police wagon" "paddy wagon" "patrol wagon" "wagon" "black Maria"
"iPod"
"punching bag" "punch bag" "punching ball" "punchball"
"beacon" "lighthouse" "beacon light" "pharos"
"jellyfish"
"wok"
"potter's wheel"
"sandal"
"pill bottle"
"butcher shop" "meat market"
```

Visualize Examples

Run the following to visualize some example images from random classses in TinyImageNet-100-A. It selects classes and images randomly, so you can run it several times to see different images.

In [4]:
```
# Visualize some examples of the training data
classes_to_show = 7
examples_per_class = 5

class_idxs = np.random.choice(len(class_names), size=classes_to_show, replace=False)
for i, class_idx in enumerate(class_idxs):
    train_idxs, = np.nonzero(y_train == class_idx)
    train_idxs = np.random.choice(train_idxs, size=examples_per_class, replace=False)
    for j, train_idx in enumerate(train_idxs):
        img = X_train[train_idx] + mean_img
        img = img.transpose(1, 2, 0).astype('uint8')
        plt.subplot(examples_per_class, classes_to_show, 1 + i + classes_to_show * j)
        if j == 0:
            plt.title(class_names[class_idx][0])
        plt.imshow(img)
        plt.gca().axis('off')

plt.show()
```



Test human performance

Run the following to test your own classification performance on the TinyImageNet-100-A dataset.

You can run several times in 'training' mode to get familiar with the task; once you are ready to test yourself, switch the mode to `'val'`.

You won't be penalized if you don't correctly classify all the images, but you should still try your best.

In [10]:
```
mode = 'train'

name_to_label = {n.lower(): i for i, ns in enumerate(class_names) for n in ns}

if mode == 'train':
    X, y = X_train, y_train
elif mode == 'val':
    X, y = X_val, y_val

num_correct = 0
num_images = 0
for i in xrange(num_images):
    idx = np.random.randint(X.shape[0])
    img = (X[idx] + mean_img).transpose(1, 2, 0).astype('uint8')
    plt.imshow(img)
    plt.gca().axis('off')
    plt.gcf().set_size_inches((2, 2))
    plt.show()
    got_name = False
    while not got_name:
        name = raw_input('Guess the class for the above image (%d / %d) : ' % (i + 1, num_images))
        name = name.lower()
        got_name = name in name_to_label
        if not got_name:
            print 'That is not a valid class name; try again'
    guess = name_to_label[name]
    if guess == y[idx]:
        num_correct += 1
        print 'Correct!'
    else:
        print 'Incorrect; it was actually %r' % class_names[y[idx]]

#acc = float(num_correct) / num_images
#print 'You got %d / %d correct for an accuracy of %f' % (num_correct, num_images, acc)
```

Download pretrained models

We have provided 10 pretrained ConvNets for the TinyImageNet-100-A dataset. Each of these models is a five-layer ConvNet with the architecture

[conv - relu - pool] x 3 - affine - relu - affine - softmax

All convolutional layers are 3x3 with stride 1 and all pooling layers are 2x2 with stride 2. The first two convolutional layers have 32 filters each, and the third convolutional layer has 64 filters. The hidden affine layer has 512 neurons. You can run the forward and backward pass for these five layer convnets using the function `five_layer_convnet` in the file `cs231n/classifiers/convnet.py`.

Each of these models was trained for 25 epochs over the TinyImageNet-100-A training data with a batch size of 50 and with dropout on the hidden affine layer. Each model was trained using slightly different values for the learning rate, regularization, and dropout probability.

To download the pretrained models, go into the `cs231n/datasets` directory and run the `get_pretrained_models.sh` script. Once you have done so, run the following to load the pretrained models into memory.

NOTE: The pretrained models will take about 245MB of disk space.

In [5]:
```
from cs231n.data_utils import load_models

models_dir = 'cs231n/datasets/tiny-100-A-pretrained'

# models is a dictionary mappping model names to models.
# Like the previous assignment, each model is a dictionary mapping parameter
# names to parameter values.
models = load_models(models_dir)

for a, b in models.iteritems():
    print a
```

```
model9
model8
model3
model2
model1
model7
model6
model5
model4
model10
```

Run models on the validation set

To benchmark the performance of each model on its own, we will use each model to make predictions on the validation set.

In [6]:
```
from cs231n.classifiers.convnet import five_layer_convnet

# Dictionary mapping model names to their predicted class probabilities on the
# validation set. model_to_probs[model_name] is an array of shape (N_val, 100)
# where model_to_probs[model_name][i, j] = p indicates that models[model_name]
# predicts that X_val[i] has class i with probability p.
model_to_probs = {}

################################################################################
# TODO: Use each model to predict classification probabilities for all images  #
# in the validation set. Store the predicted probabilities in the              #
# model_to_probs dictionary as above. To compute forward passes and compute    #
# probabilities, use the function five_layer_convnet in the file               #
# cs231n/classifiers/convnet.py.                                               #
#                                                                              #
# HINT: Trying to predict on the entire validation set all at once will use a  #
# ton of memory, so you should break the validation set into batches and run   #
# each batch through each model separately.                                    #
################################################################################

#This code worked on my machine- I didn't have to break up the validation set
for name, model in models.iteritems():
    probs = five_layer_convnet(X_val, models[name], return_probs=True)
    model_to_probs[name] = probs
    print "Finished ", name


################################################################################
#                              END OF YOUR CODE                                #
################################################################################

# Compute and print the accuracy for each model.
for model_name, probs in model_to_probs.iteritems():
    acc = np.mean(np.argmax(probs, axis=1) == y_val)
    print '%s got accuracy %f' % (model_name, acc)
```

```
Finished  model9
Finished  model8
Finished  model3
Finished  model2
Finished  model1
Finished  model7
Finished  model6
Finished  model5
Finished  model4
Finished  model10
model9 got accuracy 0.358400
model8 got accuracy 0.357000
model3 got accuracy 0.370600
model2 got accuracy 0.371000
model1 got accuracy 0.371800
model7 got accuracy 0.360000
model6 got accuracy 0.363400
model5 got accuracy 0.368600
model4 got accuracy 0.369200
model10 got accuracy 0.357000
```

Use a model ensemble

A simple way to implement an ensemble of models is to average the predicted probabilites for each model in the ensemble.

More concretely, suppose we have models k models $m_1, \ldots, m_k$ and we want to combine them into an ensemble. If $p(x = y_i \mid m_j)$ is the probability that the input x is classified as y_i under model m_j, then the enemble predicts

$$p(x = y_i \mid \{m_1, \ldots, m_k\}) = \frac{1}{k} \sum_{j=1}^{k} p(x = y_i \mid m_j)$$

In the cell below, implement this simple ensemble method by filling in the `compute_ensemble_preds` function. The ensemble of all 10 models should perform much better than the best individual model.

In [7]:
```
def compute_ensemble_preds(probs_list):
    """
    Use the predicted class probabilities from different models to implement
    the ensembling method described above.

    Inputs:
    - probs_list: A list of numpy arrays, where each gives the predicted class
      probabilities under some model. In other words,
      probs_list[j][i, c] = p means that the jth model in the ensemble thinks
      that the ith data point has class c with probability p.

    Returns:
    An array y_pred_ensemble of ensembled predictions, such that
    y_pred_ensemble[i] = c means that ensemble predicts that the ith data point
    is predicted to have class c.
    """
    y_pred_ensemble = None
    ###########################################################################
    # TODO: Implement this function. Store the ensemble predictions in        #
    # y_pred_ensemble.                                                        #
    ###########################################################################
    y_pred_ensemble = np.argmax(sum(probs_list), axis=1)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return y_pred_ensemble


# Combine all models into an ensemble and make predictions on the validation set.
# This should be significantly better than the best individual model.
print np.mean(compute_ensemble_preds(model_to_probs.values())  == y_val)
```

```
0.4244
```

Ensemble size vs Performance

Using our 10 pretrained models, we can form many different ensembles of different sizes. More precisely, if we have n models and we want to form an ensemble of k models, then there are $\binom{n}{k}$ possible ensembles that we can form, where

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

We can use these different possible ensembles to study the effect of ensemble size on ensemble performance.

In the cell below, compute the validation set accuracy of all possible ensembles of our 10 pretrained models. Produce a scatter plot with "ensemble size" on the horizontal axis and "validation set accuracy" on the vertical axis. Your plot should have a total of

$$\sum_{k=1}^{10} \binom{10}{k}$$

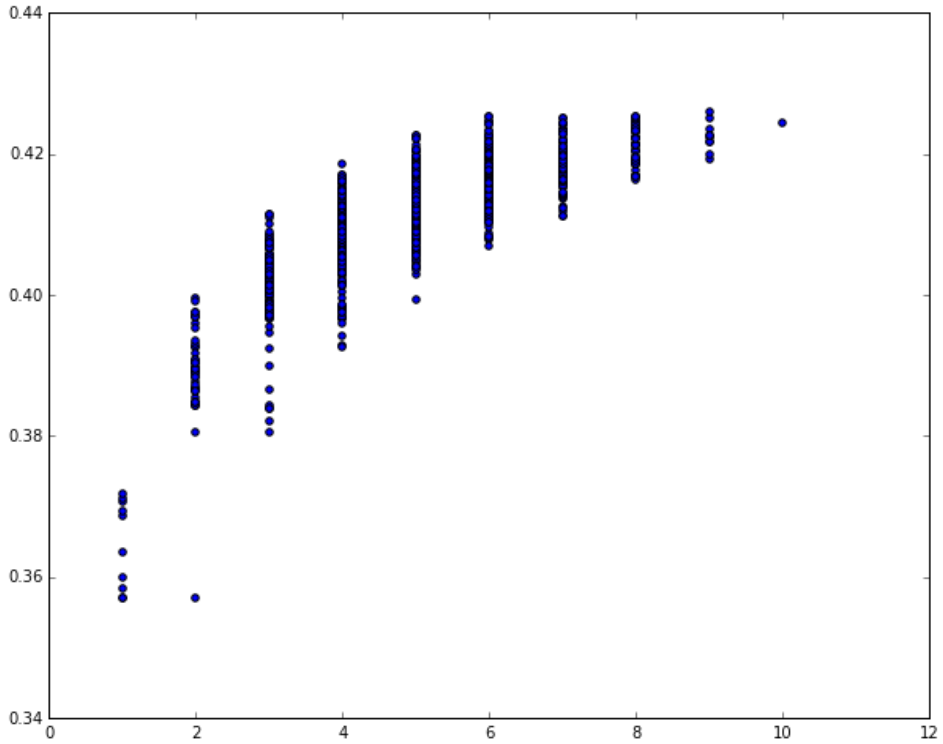
points corresponding to all possible ensembles of the 10 pretrained models.

You should be able to compute the validation set predictions of these ensembles without computing any more forward passes through any of the networks.

In [8]:
```
################################################################################
# TODO: Create a plot comparing ensemble size with ensemble performance as     #
# described above.                                                             #
#                                                                              #
# HINT: Look up the function itertools.combinations.                           #
################################################################################
import matplotlib.pyplot as plt
import itertools as it

#I got this recipe from https://docs.python.org/2/library/itertools.html
def powerset_no_null(iterable):
    s = list(iterable)
    return it.chain.from_iterable(it.combinations(s, r) for r in range(1, len(s) + 1))

models_list = model_to_probs.values()

list_of_sets = list(powerset_no_null(models_list))

print len(list_of_sets)
num_models = map(len, list_of_sets)
score = map(lambda s: np.mean(compute_ensemble_preds(s)  == y_val), list_of_sets)

plt.scatter(num_models, score)

################################################################################
#                            END OF YOUR CODE                                  #
################################################################################
```

```
1023
```

Out[8]: <matplotlib.collections.PathCollection at 0x115dadad0>



In [ ]: