q4

January 11, 2016

1 Visualizing and Breaking ConvNets

In this exercise we will visualize saliency maps for individual images and we will construct images to fool a trained ConvNet.

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
```

2 Load the data and pretrained model

You should have already downloaded the TinyImageNet-100-A dataset and the pretrained models.

In [2]:
```
# Load the TinyImageNet-100-A dataset and a pretrained model

from cs231n.data_utils import load_tiny_imagenet, load_models

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

# Load a pretrained model; it is a five layer convnet.
models_dir = 'cs231n/datasets/tiny-100-A-pretrained'
model = load_models(models_dir)['model1']
```

```
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
```

3 Compute predictions on validation set

For the experiments in this exercise it will be useful to have access to the predictions of the trained ConvNet on the TinyImageNet-100-A validation set.

In [4]:
```
from cs231n.classifiers.convnet import five_layer_convnet

# Array of shape (X_val.shape[0],) storing predictions on the validation set.
# y_val_pred[i] = c indicates that the model predicts that X_val[i] has label c.
y_val_pred = None

print X_val.shape
################################################################################
# TODO: Use the pretrained model stored in model to compute predictions on the #
# validation set. Store the results in y_val_pred.                             #
#                                                                              #
# HINT: As in the previous exercises, you will want to break the validation    #
# set into batches.                                                            #
################################################################################
y_probs = five_layer_convnet(X_val, model, return_probs=True)
y_val_pred = np.argmax(y_probs, axis=1)
print y_val_pred[:30]
################################################################################
#                                END OF YOUR CODE                              #
################################################################################

correct_indices, = np.nonzero(y_val_pred == y_val)
print len(correct_indices)
```

```
(5000, 3, 64, 64)
[33 12 15 33 97 34  2 27  5 30  2 90 28  4  5 10 13 95 14 37 21 22 60 61 76
 66  4 27 97 13]
1859
```

4 Visualize Saliency Maps

In a recent paper [1], it was suggested that you can understand which part of an image is important for classification by visualizing the gradient of the correct class score with respect to the input image. This was covered in lecture on 2/2/2015 under the section "Visualize the data gradient". Recall that if a region of the image has a high data gradient, then this indicates that the output of the ConvNet is sensitive to perturbations in that region of the input image.

We will do something similar, instead visualizing the gradient of the data loss with respect to the input image; this gives similar results and is cleaner to implement using our codebase.

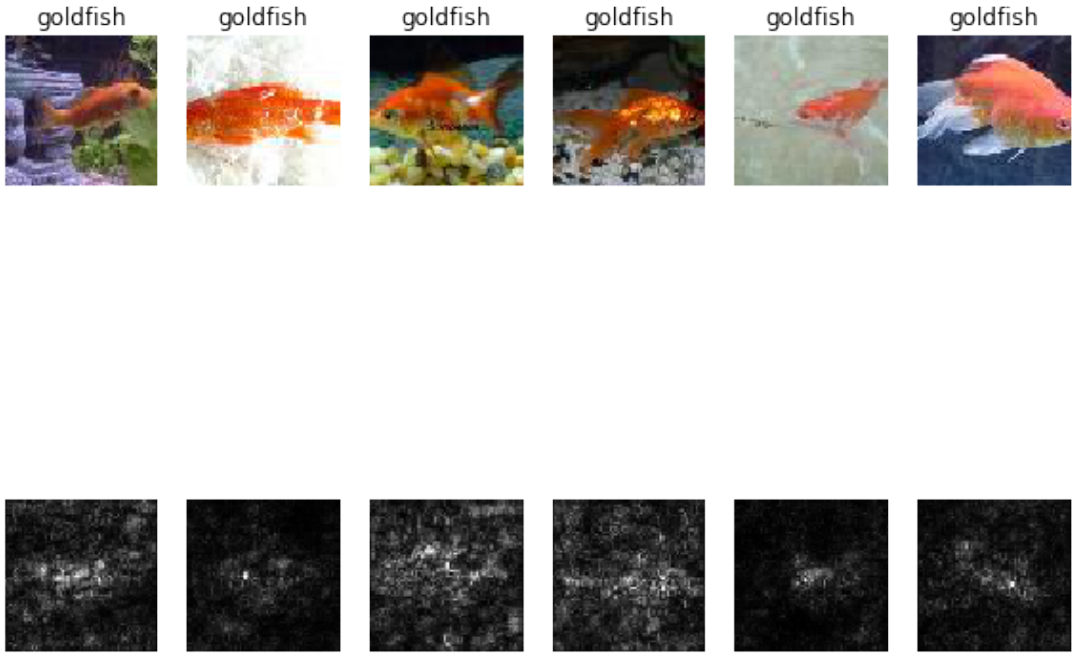
First, open the file `cs231n/classifiers/convnet.py` and modify the `five_layer_net` function to return the gradient of the loss with respect to the input when the `compute_dX` flag is true.

Once you have done so, complete the implementation in the following cell to allow you to visualize image-specific class saliency maps for images in the TinyImageNet-100-A validation set.

[1] K. Simonyan, A. Vedaldi, A. Zisserman , “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014

In [5]:
```
from cs231n.classifiers.convnet import five_layer_convnet

def show_image(img, rescale=False, add_mean=True):
    """
    Utility to show an image. In our ConvNets, images are 3D slices of 4D
    volumes; to visualize them we need to squeeze out the extra dimension,
    flip the axes so that channels are last, add the mean image, convert to
    uint8, and possibly rescale to be between 0 and 255. To make figures
    prettier we also need to suppress the axis labels after imshow.

    Input:
    - img: (1, C, H, W) or (C, H, W) or (1, H, W) or (H, W) giving
      pixel data for an image.
    - rescale: If true rescale the data to fit between 0 and 255
    - add_mean: If true add the training data mean image
    """
    img = img.copy()
    if add_mean:
        img += mean_img
    img = img.squeeze()
    if img.ndim == 3:
        img = img.transpose(1, 2, 0)
    if rescale:
        low, high = np.min(img), np.max(img)
        img = 255.0 * (img - low) / (high - low)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# The number of example images to show. You can change this.
num_examples = 6

# The label of the class to visualize. You can change this.
class_idx = 22 # goldfish

# An array of shape (num_examples,) containing the indices of validation set
# images for which saliency maps will be visualized. We wil visualize several
# examples of images from the validation set whose label is class_idx and which
# are correctly classified using the pretrained ConvNet. In other words, if
# example_idxs[i] = j then we should have y_val[j] = class_idx and the pretrained
# ConvNet should correctly classify X_val[j].
example_idxs = None

################################################################################
# TODO: Choose several examples from the validation set whose correct label is #
# class_idx and which are correctly classified by the pretrained ConvNet.      #
# Store the results in the example_idxs variable.                              #
################################################################################
class_indices = np.nonzero(y_val == class_idx)
example_idxs = np.intersect1d(correct_indices, class_indices)[:num_examples]
################################################################################
#                                END OF YOUR CODE                              #
```

```
##############################################################################

# Array to store gradients of the loss with respect to your chosen example images.
dX = np.zeros((num_examples, 3, 64, 64))

##############################################################################
# TODO: Compute image gradients for your chosen examples. Store the result in  #
# the dX variable.                                                             #
##############################################################################
dX = five_layer_convnet(X_val[example_idxs], model, y=y_val[example_idxs],
                        compute_dX=True)
##############################################################################
#                               END OF YOUR CODE                             #
##############################################################################

# Plot the images and their saliency maps.
for i in xrange(num_examples):
    # Visualize the image
    plt.subplot(2, num_examples, i + 1)
    show_image(X_val[example_idxs[i]])
    plt.title(class_names[y_val[example_idxs[i]]][0])

    # Saliency map for the ith example image.
    sal = np.zeros((64, 64))

    ##########################################################################
    # TODO: Compute the saliency map for the ith example image. Use image    #
    # derivatives from dX[i] to compute the saliency map for                 #
    # X_val[example_idxs[i]]. Store the result in the sal variable.          #
    ##########################################################################
    sal = np.amax(abs(dX[i]), axis=0)
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################

    # Visualize its saliency map.
    plt.subplot(2, num_examples, num_examples + i + 1)
    show_image(sal, rescale=True, add_mean=False)
```



5 Fooling images for ConvNets

Two other papers [1, 2] discussed in lecture on 2/2 presented the idea of performing optimization over the input images to construct images that "fool" a trained ConvNet. This paper showed that given a trained ConvNet, an input image, and a desired label, that we can add a small amount of noise to the input image to force the ConvNet to classify it as having the desired label.

In this section we will reproduce some of these results.

Suppose that $L(x, y, m)$ is the data loss under model m, where we tell the network that the input x should be classified as having label y. Given a starting image x_0, a desired label y, and a pretrained model m, we will create a fooling image x_f by solving the following optimization problem:

$$x_f = \arg\min_x \left(L(x, y, m) + \frac{\lambda}{2} \|x - x_0\|_2^2 \right)$$

The term $\|x - x_0\|^2$ is L_2 regularization in image space which encourages the fooling image to look similar to the starting image, and the constant λ is the strength of this regularization. We will use gradient descent to perform optimization under this model.

In the past, when using gradient descent we have stopped after a fixed number of iterations. Here we will use a different stopping criteria. Suppose that $p(x = y \mid m)$ is the probability that the input x is assigned the label y under the model m. We will specify a desired *confidence threshold* t for the fooling image, and we will stop our optimization when we have $p(x_f = y \mid m) >= t$.

[1] Szegedy, Christian, et al. "Intriguing properties of neural networks." arXiv preprint, 2013. [2] Nguyen, Anh, Jason Yosinski, and Jeff Clune. "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images." arXiv preprint, 2014.

```
In [6]: def make_fooling_image(img, y, model, reg=0.0, step_size=500, confidence=0.5):
            """
            Perform optimization in image space to create an image that is similar to img
            but is classified as y by model.
```

```
    Inputs:
    - img: Array of shape (1, C, H, W) containing (mean-subtracted) pixel data for
      the starting point for the fooling image.
    - y: The desired label; should be a single integer.
    - model: Dictionary mapping parameter names to weights; this is a pretrained
      five_layer_net model.
    - reg: Regularization strength (in image space) for the fooling image. This
      is the parameter lambda in the equation above.
    - step_size: The step size to use for gradient descent.
    - confidence: The desired confidence threshold for the fooling image.
    """
    fooling_img = img.copy()
    ##############################################################################
    # TODO: Use gradient descent in image space to create a fooling image,       #
    # stopping when the predicted probability for the fooling image is greater   #
    # than the specified confidence threshold.                                   #
    ##############################################################################
    while True:
        probs = five_layer_convnet(fooling_img, model, return_probs=True)
        p = probs[0][y]

        if p > confidence:
            break

        print "prob = ", p
        dX = five_layer_convnet(fooling_img, model, y, compute_dX = True)
        fooling_img -= step_size*dX

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

    return fooling_img
```

6 Fooling images from correctly classified images

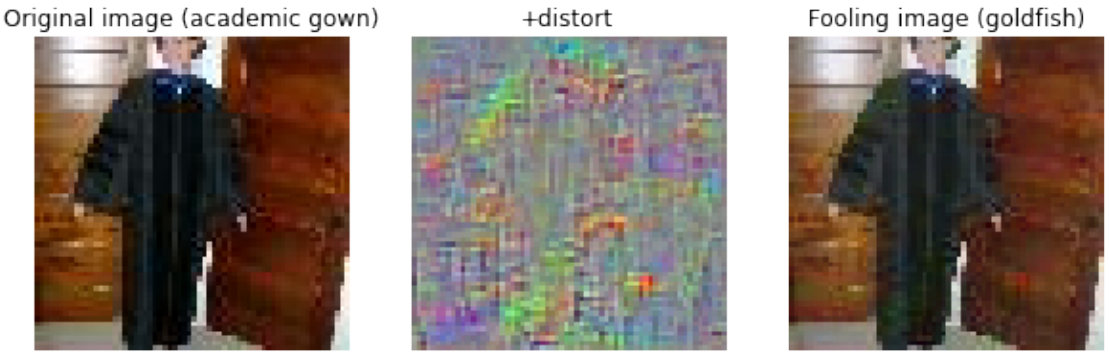
We will choose an image that is correctly classified by the pretrained network and create a fooling image that the network classifies as a goldfish.

You should experiment with different step sizes, regularizations, confidence thresholds, and target classes

In [7]:
```
# Choose a random image that is correctly classified
from cs231n.classifiers.convnet import five_layer_convnet
idx = np.random.choice(np.nonzero(y_val_pred == y_val)[0])
img = X_val[idx:idx+1]
class_idx = 22 # Goldfish
confidence = 0.5
fooling_img = make_fooling_image(img, class_idx, model, step_size=1000, reg=0.00002, confidence=

# Check that the fooling image has probability above the threshold.
assert five_layer_convnet(fooling_img, model, return_probs=True)[0, class_idx] >= confidence, \
       'The ConvNet is not fooled.'

# Show the original image
```

```
        plt.subplot(1, 3, 1)
        plt.title('Original image (%s)' % class_names[y_val[idx]][0])
        show_image(img)

        # Show the difference between the original and fooling image
        plt.subplot(1, 3, 2)
        plt.title('+distort')
        show_image(fooling_img - img, add_mean=False, rescale=True)

        # Show the fooling image
        plt.subplot(1, 3, 3)
        plt.title('Fooling image (%s)' % class_names[class_idx][0])
        show_image(fooling_img, rescale=True)
```

```
prob =  1.08676e-09
prob =  4.05834e-08
prob =  6.31808e-07
prob =  4.88433e-06
prob =  2.15566e-05
prob =  7.3939e-05
prob =  0.000215442
prob =  0.000573549
prob =  0.00130637
prob =  0.00268248
prob =  0.00515045
prob =  0.00842242
prob =  0.0124042
prob =  0.0173929
prob =  0.0239796
prob =  0.032787
prob =  0.0443235
prob =  0.058301
prob =  0.0753929
prob =  0.0950768
prob =  0.117006
prob =  0.14179
prob =  0.166041
prob =  0.191667
prob =  0.219531
prob =  0.24595
prob =  0.271083
prob =  0.293795
prob =  0.315811
prob =  0.337317
prob =  0.358934
prob =  0.381381
prob =  0.402909
prob =  0.423955
prob =  0.44418
prob =  0.46332
prob =  0.481365
prob =  0.498882
```



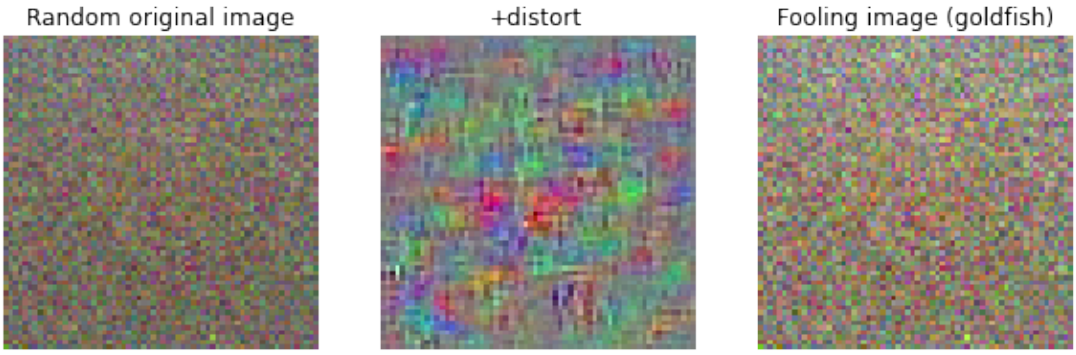
7 Fooling image from random noise

Instead of starting from a correctly classified image, we can instead start our optimization from random noise. This will allow us to produce fooling images that do not look like anything to humans.

You should experiment with the scale of the initial random noise, the step size, the regularization, the confidence threshold, and the target class.

```
In [8]: # Generate random noise to start
        img = 20 * np.random.randn(1, 3, 64, 64)
        class_idx = 22 # Goldfish
        fooling_img = make_fooling_image(img, class_idx, model, step_size=500, reg=0.00005, confidence=

        # Check that the fooling image has probability above the threshold.
        assert five_layer_convnet(fooling_img, model, return_probs=True)[0, class_idx] >= confidence, \
               'The ConvNet is not fooled.'

        # Show the original image
        plt.subplot(1, 3, 1)
        plt.title('Random original image')
        show_image(img)

        # Show the difference between the original and fooling image
        plt.subplot(1, 3, 2)
        plt.title('+distort')
        show_image(fooling_img - img, add_mean=False, rescale=True)

        # Show the fooling image
        plt.subplot(1, 3, 3)
        plt.title('Fooling image (%s)' % class_names[class_idx][0])
        show_image(fooling_img, rescale=True)

prob =  0.0213712408931
prob =  0.032154104041
prob =  0.0457846777966
prob =  0.067916425721
prob =  0.0994408501256
prob =  0.135759949871
prob =  0.17737367418
```

```
prob =  0.22550977309
prob =  0.27423378365
prob =  0.325392539108
prob =  0.379276635658
prob =  0.42707220024
prob =  0.468967438975
```




```
In [9]: probs = five_layer_convnet(fooling_img, model, return_probs=True)
        print probs
```

```
[[  1.71990188e-04   1.08053246e-03   5.77410761e-05   3.71215124e-05
    8.67506848e-03   4.17582305e-03   7.04728541e-05   2.63942215e-03
    2.17370122e-04   4.38525433e-04   3.55103584e-04   1.27094357e-04
    1.02657076e-05   8.24690981e-02   2.69452526e-04   2.36236759e-04
    1.63840415e-03   3.81819114e-03   5.21477494e-03   1.62656821e-06
    5.84551743e-05   7.54797300e-04   5.08649536e-01   4.64544469e-04
    1.48661014e-05   1.00662758e-04   6.34862792e-05   1.44710059e-04
    5.00177085e-03   6.27719786e-04   9.31599498e-05   2.04709741e-03
    9.37878184e-05   1.87731547e-02   1.68848220e-02   3.80743742e-04
    1.19075383e-02   1.22667325e-01   1.23027997e-04   1.44756705e-04
    3.71399222e-03   3.53003882e-04   1.82124083e-02   6.83190556e-04
    2.42695072e-03   1.99897489e-03   8.94671911e-03   1.30338708e-02
    1.03188268e-02   2.78015824e-04   7.34644007e-04   5.76247741e-03
    1.47677488e-05   8.86527980e-03   1.53315732e-04   1.42677726e-03
    2.65324939e-03   1.92923589e-05   6.94589334e-06   3.20409512e-05
    4.27668283e-04   3.23358139e-04   5.09968822e-04   3.39538719e-03
    4.17862337e-05   7.01461664e-04   2.37931072e-04   5.95208536e-04
    2.81065347e-04   6.91591730e-06   5.72175115e-06   6.78666717e-04
    5.92818834e-03   7.28040648e-04   8.38110757e-06   2.26732457e-03
    4.76524185e-02   1.41168138e-04   1.44203848e-03   1.64276107e-06
    3.08615520e-05   9.63194692e-05   3.05911877e-03   5.54253239e-05
    2.22599223e-03   1.00635843e-06   4.32001838e-04   6.12181749e-03
    2.44634622e-04   3.42736859e-02   1.23859306e-05   4.64595320e-05
    3.22634129e-04   1.18003694e-04   2.64182820e-03   1.02266140e-03
    2.72806360e-05   1.84396783e-03   1.56868451e-03   1.45867925e-04]]
```

```
In [ ]:
```