Network Visualization (11 Points)

In the first part of the notebook we will examine the use of different types of attribution algorithms - both gradient and perturbation - for images, and understand the differences using the Tiny Imagenet dataset and a pretrained SqueezeNet model.

Links to the PyTorch, Tiny Imagenet, and to the SqueezeNet paper: https://pytorch.org/docs/stable/index.html
https://tiny-imagenet.herokuapp.com/ https://arxiv.org/pdf/1602.07360.pdf

Tiny Imagenet: Andrej Karpathy and Jeffrey Johnson, "Training Imagenet on Tiny Imagenet", Stanford University, Visualizing Image Classification Models and Saliency Maps", ICLR Workshop, 2016.
Gradient Saliency: Karen Simonyan, Andrea Vedaldi, Andrew Zisserman, "Deep Networks" (2014).
Saliency Maps: Karen Simonyan, "Exploiting Saliency in Convolutional Neural Networks", Visualizing and Understanding Convolutional Networks", ICLR Workshop, 2014.
Fooling Images: Dumitru Erhan, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, Dhruv Batra, "Gradient Adversarial Training of Neural Networks", Machine Learning, Computer Vision, 2015.

For the full list of attribution algorithms details check out: https://arxiv.org/pdf/

In this second and third parts we will focus on generating fake images, by tricking and implementing key components to a pure classifier.

Our goal is to fool a pretrained image classifier, in the case of Tiny Imagenet, into thinking that an image is a certain class.

We will start by first fooling the classifier, and then we will guide you to understand their deeper earlier problems.

After training is done, we will use the trained classifier to generate fake images, using the models performance, as that can help us understand how the parameters in the network with respect to the model parameters, and perform gradient descent on the network parameters to maximize the loss.

Unfortunately, the results are not perfectly efficient. We will see that a convolutional neural network model which has been pretrained to perform image classification on the Imagenet dataset. We will see that model is able to accurately classify a image, but a slight change in the image, imperceptible to the human eye, can cause the model to misclassify the image. We will then implement the fooling image algorithm, which will take a single image and perform a small update to the image to maximize the score of a target class.

- **Saliency Maps:** Saliency maps are a quick way to tell which part of the image influenced the classification decision made by the pretrained model.
- **Fooling Images:** We can perturb a single image to fool the model so it thinks it is any class we want.
- **Feature Inversion:** We can perturb an random image so that it appears like the feature maps for some image, but will be indistinguishable by the pretrained model.
- **Adversarial Attacks:** We can perturb an image so that it appears like a different image to the human eye, but the pretrained model will still classify the image as the original class we started with.

We will use **PyTorch** in these first three parts of this notebook, which has been installed with Python3.6 on your virtual files.

Make sure you login with your Stanford email, to access the Tiny Imagenet dataset.

Install Imagenet and extract images as the pretrained model, in the terminal:
```
wget -nc http://cs231n.stanford.edu/tiny-imagenet.zip
unzip -nq tiny-imagenet.zip
mv -n tiny-imagenet-200 imagenet
```

```
[1]: ! wget -nc
```

```
[2]: ! unzip -qq
```

Helper Functions

Our pretrained model also requires to have their images preprocessed by subtracting the per-color mean and dividing by the per-color standard deviation. We define a few helper functions for performing and undoing this preprocessing.

You don't need to do anything in this cell.

```
[3]: # Imports
import numpy as np
import os, h5py, sys, argparse
import matplotlib.pyplot as plt
from cs231n.gan_solver import PytorchNet
from cs231n.data_utils import load_tiny_imagenet, load_models
from cs231n.image_utils import jitter, blur_image
from cs231n.vis_utils import rescale, tile_grid_image_from_ndarray
from cs231n.fooling_utils import jitter_gradients, clip_image_values
from collections import OrderedDict
from scipy.ndimage import gaussian_filter1d, gaussian_filter
from scipy.misc import imresize

# Utility for saving figures
def save(name, fig, kwargs):
    if not os.path.isdir('images'):
        os.mkdir('images')
    fn = lambda ext, name=name, fig=fig, kwargs=kwargs: os.path.join('images', name + '.' + ext)
    print('Saving figure to %r...' % fn('*', **kwargs))
    for ext in ['png', 'pdf']:
        print('%r...' % ext)
        fig.savefig(fn(ext, **kwargs), **kwargs)
    print('Done.')

%matplotlib inline
plt.rcParams['figure.figure.dpi'] = 72 # for high dpi displays
plt.rcParams['figure.subplot.wspace'] = 0.1
plt.rcParams['figure.subplot.hspace'] = 0.1
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['image.cmap'] = 'gray'
plt.rcParams['savefig.bbox'] = 'tight'
plt.rcParams['savefig.pad_inches'] = 0.1
plt.rcParams['savefig.dpi'] = 72
```

```
[4]: %load_ext autoreload
```

Helper Functions

Our pretrained model also requires to have their images preprocessed by subtracting the per-color mean and dividing by the per-color standard deviation. We define a few helper functions for performing and undoing this preprocessing.

You don't need to do anything in this cell.

```
[5]: # Load the Tiny Imagenet validation set
data = load_tiny_imagenet('imagenet/tiny-imagenet-100-A', subtract_mean=True)

# Load the PyTorch SqueezeNet model
model = PytorchNet(weight_dir='imagenet', single_output=True)

# Prepare the model for validation
for param in model.parrameters():
    param.requires_grad = False
model.eval()
model.cuda()

# Extra layer to blur feature maps
blur = lambda sigma: gaussian_filter1d(sigma, axis=2)

def deprocess(img, rescale=False):
    return rescale(img.transpose(1, 2, 3, 0).transpose(0, 3, 1, 2).copy() if rescale else img.transpose(1, 2, 3, 0).transpose(0, 3, 1, 2)[[2, 1, 0]].copy()
[6]: ! ls imagenet
```

```
[7]: ! ls imagenet/tiny-imagenet-100-A
```

Pretrained Model

For all of our image gradient experiments, we will start with a convolutional neural network which was pretrained to perform image classification on ImageNet. We can use this model in PyTorch; and the weights of the pretrained model are in SqueezeNet. We'llwrap pretrained model in a PyTorch wrapper to take numpy arrays and output numpy arrays.

Using SqueezeNet rather than AlexNet or VGG or ResNet means that we can easily perform image gradient experiments on a pretrained model that doesn't take too much memory. By the end of the notebook you will be ablle to visualize AlexNet feature maps and vis pretrained VGG and ResNet models on your own pretrained image.

Pretrained models were ported to PyTorch from Caffe using the tool https://github.com/ksimonyan/caffe-public/tree/master/tools/extra, and ported to TensorFlow using the tool https://github.com/ethereon/caffe-tensorflow

- Models ported to TensorFlow: AlexNet and VGG-16 (TensorFlow weights are in TensorFlow-HDF5 model zoo)

```
[8]: # Download and unpack the pretrained model and Tiny Imagenet validation data
!wget -nc -q http://cs231n.stanford.edu/pytorch-models.zip
!unzip -qq -o pytorch-models.zip
```

Load some Imagenet images

If you have trouble accessing the downloading script, there is a minimal set of images that you have to do it now. We have provided a few example images from the validation set of the Imagenet ILSVRC-2012 Classification dataset.

You downloaded these images in:
Tiny Imagenet Dataset: imagenet/tiny-imagenet-100-A

We will use, throughout the paper, to name a variable:

X_validation stands for the images in the validation set.
y_validation stands for the labels in the validation set, which are the ground truth classes for the images.
y_pred stands for the predicted classes for the images, using the pretrained model, and the images in X_validation.
X_fooling stands for the images that we have perturbed to fool the pretrained model into classifying them as the class in y_fooling.
y_fooling stands for the classes we want the pretrained model to classify the images in X_fooling.
X_fooling_pred stands for the classes the pretrained model actually classifies the images in X_fooling.

```
[9]: # Load the validation data
X_validation, y_validation = data['X_val'], data['y_val']
[10]: ! ls imagenet/tiny-imagenet-100-A/val/n*
```

```
[11]: # Print the shapes of the validation data
print('X_validation shape: ', X_validation.shape)
print('y_validation shape: ', y_validation.shape)
```

```
X_validation shape:  (5000, 3, 64, 64)
y_validation shape:  (5000,)
```

```
[12]: # Load the class names
class_to_name = data['class_to_name']
print(class_to_name)
```

```
{0: 'n02074367', 1: 'n02085620', 2: 'n02099601', 3: 'n02099712', 4: 'n02106662', 5: 'n02113799', 6: 'n02123394', 7: 'n02129165', 8: 'n02132136', 9: 'n02165456', 10: 'n02279972', 11: 'n02317335', 12: 'n02342885', 13: 'n02363005', 14: 'n02364673', 15: 'n02391049', 16: 'n02410509', 17: 'n02423022', 18: 'n02444819', 19: 'n02480495', 20: 'n02509815', 21: 'n02666347', 22: 'n02669723', 23: 'n02699494', 24: 'n02730930', 25: 'n02769748', 26: 'n02788148', 27: 'n02791270', 28: 'n02793495', 29: 'n02837789', 30: 'n02841315', 31: 'n02905151', 32: 'n02908123', 33: 'n02909870', 34: 'n02917067', 35: 'n02927161', 36: 'n02948072', 37: 'n02950826', 38: 'n02963159', 39: 'n02977058', 40: 'n02988304', 41: 'n03026506', 42: 'n03042490', 43: 'n03085013', 44: 'n03100240', 45: 'n03126707', 46: 'n03160309', 47: 'n03179701', 48: 'n03201208', 49: 'n03250847', 50: 'n03255030', 51: 'n03355925', 52: 'n03388043', 53: 'n03393912', 54: 'n03400231', 55: 'n03404251', 56: 'n03424325', 57: 'n03444034', 58: 'n03447447', 59: 'n03544143', 60: 'n03584254', 61: 'n03599486', 62: 'n03617480', 63: 'n03626115', 64: 'n03635032', 65: 'n03670208', 66: 'n03706229', 67: 'n03733131', 68: 'n03804744', 69: 'n03814639', 70: 'n03837869', 71: 'n03854065', 72: 'n03929660', 73: 'n03930313', 74: 'n03937543', 75: 'n03970156', 76: 'n03976657', 77: 'n03977966', 78: 'n03980874', 79: 'n03983396', 80: 'n04074963', 81: 'n04085374', 82: 'n04099969', 83: 'n04118538', 84: 'n04133789', 85: 'n04146614', 86: 'n04149813', 87: 'n04252077', 88: 'n04254680', 89: 'n04259630', 90: 'n04265275', 91: 'n04270147', 92: 'n04286575', 93: 'n04330267', 94: 'n04356056', 95: 'n04371430', 96: 'n04376876', 97: 'n04398044', 98: 'n04399382', 99: 'n04507155'}
```

```
[13]: # Vanilla frozen SqueezeNet model
model = PytorchNet(weight_dir='imagenet', single_output=True)
for param in model.parrameters():
    param.requires_grad = False
model.eval()
model.cuda()
```

```
[14]: # Preprocessed validation data
X_validation = data['X_val']
y_validation = data['y_val']
```

Saliency Maps (2 pts)

Using this pretrained model, we will compute class saliency maps as described in the paper:

Efficient Estimation of Word Representations in Vector Space, Tomas Mikolov Corrado Kai Chen Greg Dean Jeffrey Dean, Distributed Representations of Words and Phrases and their Compositionality, Tomas Mikolov Ilya Sutskever Kai Chen Greg Corrado

Let's try attribute to a selected layer and visualize the attribution for any selected channel. We can choose to change the layers and channels and observe the attribution changes.

Please read the Feature Visualization https://distill.pub/2017/feature-visualization/#models-optimization-objectives and Attribution Spatial Attribution https://distill.pub/2019/activation-atlas/#spatial-attribution for detailed explanations.

Please, run the cells below, the training initialization and the new plot visualizer those using the technique described in the previous section.

```
In [ ]: from captum.attr import IntegratedGradients, LayerConductance, LayerActivation

# Initialize the attribution algorithm with the trained model.
# We are interested in the last layer.
ig = IntegratedGradients(net)

layer_cond = LayerConductance(net, net.inception)
layer_act = LayerActivation(net, net.inception)
layer_attr_dim = layer_cond.attribute(test_input, target=labels).shape
print('Layer Conductance shape: {}'.format(layer_attr_dim))
# Preparing the baseline and assigning some random values to the image.
# We will use this baseline to see how the attribution changes.
rand_img_dist = torch.cat([test_input * 0, test_input * 1])
# Defining the number of steps for the approximation.
n_steps = 200

inps, delts, labs = torch.cat([test_input, test_input]), torch.zeros(*layer_attr_dim), torch.cat([labels, labels])
# Computing the attribution.
layer_cond_vals = layer_cond.attribute(rand_img_dist, target=labels, n_steps=n_steps)
# Computing the activation.
layer_act_vals = layer_act.attribute(rand_img_dist)
print('Layer Conductance shape: {}'.format(layer_cond_vals.shape))
print('Layer Activation shape: {}'.format(layer_act_vals.shape))

# Imports for visualization.
import numpy as np
import matplotlib.pyplot as plt, matplotlib.cm as cm
from matplotlib.colors import LinearSegmentedColormap

# Defining custom colormaps.
default_cmap = LinearSegmentedColormap('GreenPurple', {'red': [(0.0, 0.0, 0.0),
                                                               (1.0, 0.0, 0.0)],
                                                       'green': [(0.0, 0.5, 0.5),
                                                                 (1.0, 0.5, 0.5)],
                                                       'blue': [(0.0, 0.0, 0.0),
                                                                (1.0, 0.0, 0.0)]},
                                       256)

# Defining a function that prepares the attributions for visualization.
def prepare_attrs(attrs, ch=0):
    r, c = attrs.shape[0], attrs.shape[1]
    return attrs[:,:,ch].view().reshape(r, c, 1)

# Defining a function that prepares the images for visualization.
def prepare_imgs(imgs):
    return imgs.view().reshape(imgs.shape[0], imgs.shape[2], imgs.shape[3], 1)

# Defining a function that plots the attributions and images.
def plot_attr(cond_vals, cond_ch, act_vals, act_ch, imgs, inds):
    fig, axs = plt.subplots(len(inds), 3, figsize=(12,4*len(inds)))
    for i, ind in enumerate(inds):
        axs[i][0].imshow(prepare_imgs(imgs[ind]).squeeze(), cmap=default_cmap)
        axs[i][1].imshow(prepare_attrs(cond_vals[ind], cond_ch).squeeze(), cmap=default_cmap)
        axs[i][2].imshow(prepare_attrs(act_vals[ind], act_ch).squeeze(), cmap=default_cmap)
    fig.tight_layout()
    return fig
```

```
Layer Conductance shape: (8, 832, 4, 4)
Layer Conductance shape: (8, 832, 4, 4)
Layer Activation shape: (8, 832, 4, 4)
```

```
In [ ]: inds = [1, 2, 3, 4, 5, 6, 7, 8]
plot_attr(layer_cond_vals, 0, layer_act_vals, 0, test_input, inds)
```

Footing Images (2 pts)

We can also use the trained model to generate adversarial examples to test the stability of the network. Consider a state-of-the-art deep neural network that classifies images with high accuracy. We'll see that small perturbations can drastically change the network's classification of the image. Surprisingly, even imperceptible non-random perturbations of a natural image can result in a different network classification.

Read the paper, and then implement the following function to generate fooling images.

```
In [ ]: def fgsm_attack(image, eps, data_grad):
    # Collect the element-wise sign of the data gradient
    sign_data_grad = data_grad.sign()
    # Delta is epsilon times the sign of the data gradient.
    delta = eps*sign_data_grad
    # Perturb the image by adding delta
    perturbed_image = image + delta
    # Return perturbed image.
    return perturbed_image
```

Now you can run the following cell to generate a fooling image. You can use the message Pocket the toilet when you succeed.

```
In [ ]: # Get a batch of test images.
test_iter = iter(test_loader)
test_input, test_labels = next(test_iter)
# Select a random one and requires_grad.
test_input = torch.unsqueeze(test_input[random.randint(0, len(test_input))], 0).requires_grad_(True)
# Calculate the output.
test_output = net(test_input)
# Calculate the loss.
test_loss = F.nll_loss(test_output, test_labels, reduction='sum')
# Zero all existing gradients.
net.zero_grad()
# Calculate gradients.
test_loss.backward()
# Collect the data gradient.
test_data_grad = test_input.grad.data
# Call FGSM Attack.
eps = 0.15
test_perturbed_input = fgsm_attack(test_input, eps, test_data_grad)
# Re-classify the perturbed image.
test_output = net(test_perturbed_input)
# Get the index of the max log-probability.
test_pred = test_output.max(1, keepmdim=True)[1]
print("Prediction: {}".format(test_pred.item()))
# Show the image.
test_input = test_input.squeeze(0).detach.cpu().numpy()
test_input = np.transpose(test_input, (1, 2, 0))
plt.imshow(test_input)
plt.show()
```

```
Prediction: 24
```

Now you can let the following cell a **generate a fooling image**. You can use the message Pocket the toilet when you succeed.

```
In [ ]: # Get a batch of test images.
test_iter = iter(test_loader)
test_input, test_labels = next(test_iter)
# Select a random one and requires_grad.
test_input = torch.unsqueeze(test_input[random.randint(0, len(test_input))], 0).requires_grad_(True)
# Calculate the output.
test_output = net(test_input)
# Calculate the loss.
test_loss = F.nll_loss(test_output, test_labels, reduction='sum')
# Zero all existing gradients.
net.zero_grad()
# Calculate gradients.
test_loss.backward()
# Collect the data gradient.
test_data_grad = test_input.grad.data
# Call FGSM Attack.
eps = 0.15
test_perturbed_input = fgsm_attack(test_input, eps, test_data_grad)
# Re-classify the perturbed image.
test_output = net(test_perturbed_input)
# Get the index of the max log-probability.
test_pred = test_output.max(1, keepmdim=True)[1]
print("Prediction: {}".format(test_pred.item()))
# Show the image.
test_input = test_input.squeeze(0).detach.cpu().numpy()
test_input = np.transpose(test_input, (1, 2, 0))
plt.imshow(test_input)
plt.show()
```

```
Prediction: 24
```

```
In [ ]: # Import the libraries.
import torchvision, torchvision.transforms, matplotlib, os
```

```
In [ ]: # Defining the transform that converts the image from numpy array to a tensor.
transform = transforms.ToTensor()
# Downloading the CIFAR-10 training set.
cifar_trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
# Defining the classes in the dataset.
cifar_classes = ('plane', 'car', 'bird', 'cat',
                 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
# Defining the number of images in the batch.
batch_size = 8
# Defining the loader.
cifar_trainloader = torch.utils.data.DataLoader(cifar_trainset, batch_size=batch_size,
                                                shuffle=True, num_workers=2)
# Obtaining random training images.
dataiter = iter(cifar_trainloader)
cifar_images, cifar_labels = dataiter.next()
# Defining the number of rows in the figure.
num_rows = 1
# Defining the figure.
fig = plt.figure(figure=(batch_size*2, num_rows*2.5))
# Looping through the images.
for i in range(batch_size):
    ax = fig.add_subplot(num_rows, batch_size/num_rows, i+1)
    ax.set_title('{}'.format(cifar_classes[cifar_labels[i]]))
    cifar_image = cifar_images[i] / 2 + 0.5
    cifar_image = np.transpose(cifar_image, (1, 2, 0))
    ax.imshow(cifar_image)
fig.tight_layout()
```

```
Files already downloaded and verified
```

Class visualization (3 pts)

Starting from a random image, optimize the image to maximize a certain score. We choose a pre-trained network and a target class, and then compute the gradient of the score function with respect to pixels of the image. We then update the image by ascending along the gradient. We can further improve the quality of the generated image by jointly optimizing for the score and for natural image statistics.

Please, read the Class visualization section of the paper and implement the following functions, such that we can visualize the pre-trained SqueezeNet model. For more details, see the paper, but the following simple regularizers will work fairly well: penalizing the norm of the image, penalizing the total variation of the image, and clipping pixels that are very close to zero (in the preprocessed image).

In the cell series, complete the implementation of the create_class_visualization function.

```
In [ ]: def jitter(X, ox, oy):
    """
    Helper function for randomly jittering an image.

    Inputs
    - X PyTorch Tensor of shape (N, C, H, W)
    - ox, oy: How many pixels to shift to the right and down
    - jitter: Randomly shifted version of image X, ox, oy
    - ox: How many pixels to shift right
    - oy: How many pixels to shift down

    Returns
    - X: Image with specified jitter.
    """
    if ox > 0:
        left = X[:, :, :, :-ox]
        right = X[:, :, :, -ox:]
        X = torch.cat([right, left], dim=3)
    if oy > 0:
        top = X[:, :, :-oy, :]
        bottom = X[:, :, -oy:, :]
        X = torch.cat([bottom, top], dim=2)
    return X
```

```
In [ ]: def create_class_visualization(target_y, model, dtype, **kwargs):
    """
    Generate an image to maximize the score of target_y under a pretrained model.

    Inputs
    - target_y: Integer in the range [0, 1000) giving the index of the class
    - model: A pretrained CNN that will be used to generate the image
    - dtype: Torch datatype to use for computing the image.

    Keyword Arguments
    - l2_reg: Strength of L2 regularization on the image
    - learning_rate: How big to make the steps in gradient ascent
    - num_jitters: How big to make jitter
    - blur_every: How often to blur the image as an implicit regularizer
    - max_jitter: How big to allow jitter
    - show_every: How often to show the intermediate result
    """
    model.type(dtype)
    l2_reg = 1e-3
    learning_rate = 25
    num_jitters = 1
    blur_every = 4
    max_jitter = 16
    show_every = 25
    for t in range(1, 201):
        if t % show_every == 0 and (verbose or t == 1):
            print('Iteration +', t)
        ox, oy = random.randint(0, max_jitter), random.randint(0, max_jitter)
        img = jitter(img, ox, oy)

        # Wrap the image in a Variable.
        img_var = Variable(img, requires_grad=True)

        # Forward pass.
        scores = model(img_var)

        # Compute the loss.
        loss = (-1 * scores[0, target_y]) + l2_reg * torch.sum(img_var ** 2)

        # Backward pass.
        loss.backward()

        # Make the step.
        img = img + learning_rate * img_var.grad

        # Clip image.
        img = jitter(img, -ox, -oy)

        # Hyperparameters for numerical stability.
        img = torch.clamp(img, -255.0, 255.0)

        # Zero the gradients.
        img_var.grad.zero_()

        # Print the image.
        img = img / 255.0
        img = img - torch.min(img)
        img = img / torch.max(img)
        img = img * 255.0

        # Show the current image.
        if t % show_every == 0 and (verbose or t == 1):
            plt.imshow(np.transpose(np.uint8(np.squeeze(img.numpy().astype(np.int32)).clip(0, 255)), (1, 2, 0)))
            plt.show()

        # Blur every 4 iterations.
        if t % blur_every == 0:
            img = torch.tensor(gaussian_filter(img.numpy(), sigma=(0, 0.5, 0.5, 0)))

        # Clip pixels that are very close to zero.
        img = img * (img > np.percentile(img.numpy(), 1))

    return img
```

```
In [ ]: # Initialize the model.
model = torchvision.models.squeezenet1.0(pretrained=True)
# Extracting the dtype.
dtype = model.classifier[1].weight.data.type()
# Let's visualize a few classes.
for class_idx, name in [(24, 'Pocket'), (1, 'goldfish'), (250, 'husky'), (309, 'bee'), (504, 'coffee mug')]:
    print('Initial image for class %d (%s)' % (class_idx, name))
    class_visualization = create_class_visualization(class_idx, model, dtype)
```

```
Initial image for class 24 (Pocket)
Initial image for class 1 (goldfish)
Initial image for class 250 (husky)
Initial image for class 309 (bee)
Initial image for class 504 (coffee mug)
```

Now you can take visualization of other classes! You should also feel free to play with various hyperparameters to try and improve the quality of the generated image, but this is not required.

```
In [ ]: # Initialize the model.
model = torchvision.models.squeezenet1.0(pretrained=True)
# Extracting the dtype.
dtype = model.classifier[1].weight.data.type()
# Let's visualize a few classes.
for class_idx, name in [(24, 'Pocket'), (1, 'goldfish'), (250, 'husky'), (309, 'bee'), (504, 'coffee mug')]:
    print('Initial image for class %d (%s)' % (class_idx, name))
    class_visualization = create_class_visualization(class_idx, model, dtype)
```

```
Initial image for class 24 (Pocket)
Initial image for class 1 (goldfish)
Initial image for class 250 (husky)
Initial image for class 309 (bee)
Initial image for class 504 (coffee mug)
```