CS 4644/7643: Deep Learning
Assignment 3

Instructor: Zsolt Kira
TAs: Ting-Yu Lan, Anshul Ahluwalia, Ahmed Shaikh, Aaditya Singh, Yash Jain, Yash Jakhotiya, Hoon Lee, Zach Minot, Sumedh Vijay, Ningyuan Yang, Jan Vijay Singh, Aaditya Singh
Discussions: https://piazza.com/gatech/spring2023/cs46447643

Deadline: 11:59 pm March 10, 2023

- Discussion is encouraged, but each student must write his/her own answers and explicitly mention any collaborators.
- Each student is expected to respect and follow the GT Honor Code: https://osi.gatech.edu/content/honor-code. **We will apply anti-cheating software to check for plagiarism**. We cross-check source code within the class, with previous years' solutions, and with online solutions. Any case that deemed substantial by the teaching team will be reported to OSI and receive a 0.
- Please **do not change the filenames and function definitions** in the skeleton code provided, as this will cause the test scripts to fail and you will receive no points in those failed tests. You may also **NOT** change the import modules in each file or import additional modules.
- It is your responsibility to make sure that all code and other deliverables are in the correct format and that your submission compiles and runs. We will not manually check your code (this is not feasible given the class size). Thus, **non-runnable code in our test environment will directly lead to a score of 0**. Also, your entire programming parts will **NOT** be graded and given 0 score if your code prints out anything that is not asked in each question.
- For the Style transfer section, you will need to run *main.ipynb* to test your code. Please append this notebook to the write-up and submit it under HW3-Writeup on Gradescope.

This assignment is designed to be run on either Google Colab or your local machine. For **Google Colab**, upload the files and then follow the Google Cloud setup directions in the notebook. For running in a **local environment**, follow the setup directions in *local_environment/SETUP.md*.

1 Network Visualization

In the first part we will explore the use of different type of attribution algorithms - both gradient and perturbation - for images, and understand their differences using the Captum model interpretability tool for PyTorch. As an exercise you'll be also asked to implement Saliency Maps from scratch.

1. Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

2. Mukund Sundararajan, Ankur Taly, Qiqi Yan, "Axiomatic Attribution for Deep Networks", ICML, 2017

3. Matthew D Zeiler, Rob Fergus, "Visualizing and Understanding Convolutional Networks", Visualizing and Understanding Convolutional Networks, 2013.

4. Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, Dhruv Batra, Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization, 2016

In the second and third parts we will focus on generating new images, by studying and implementing key components in two papers:

1. Szegedy et al, "Intriguing properties of neural networks", ICLR 2014

2. Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML 2015 Deep Learning Workshop

You will need to first read the papers, and then we will guide you to understand them deeper with some problems.

When training a model, we define a loss function which measures our current unhappiness with the model's performance; we then use backpropagation to compute the gradient of the loss with respect to the model parameters, and perform gradient descent on the model parameters to minimize the loss.

In this homework, we will do something slightly different. We will start from a convolutional neural network model which has been pretrained to perform image

classification on the ImageNet dataset. We will use this model to define a loss function which quantifies our current unhappiness with our image, then use backpropagation to compute the gradient of this loss with respect to the pixels of the image. We will then keep the model fixed, and perform gradient descent on the image to synthesize a new image which minimizes the loss.

We will explore four different techniques:

1. **Saliency Maps**: Saliency maps are a quick way to tell which part of the image influenced the classification decision made by the network.

2. **GradCAM**: GradCAM is a way to show the focus area on an image for a given label.

3. **Fooling Images**: We can perturb an input image so that it appears the same to humans, but will be misclassified by the pretrained network.

4. **Class Visualization**: We can synthesize an image to maximize the classification score of a particular class; this can give us some sense of what the network is looking for when it classifies images of that class.

We recommend using PyTorch 1.4 to finish the problems in this assignment, which has been tested with Python3.6 on Linux and Mac although PyTorch versions $\geq$ 1.0 should generally work as well.

1.1 Saliency Map

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

[1] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

A saliency map tells us the degree to which each pixel in the image affects the classification score for that image. To compute it, we compute the gradient of the unnormalized score corresponding to the correct class (which is a scalar) with respect to the pixels of the image. If the image has shape (3, H, W) then this gradient will also have shape (3, H, W); for each pixel in the image, this gradient tells us the amount by which the classification score will change if the pixel changes by a small amount. To compute the saliency map, we take the absolute value of this gradient, then take the maximum value over the 3 input channels; the final saliency map thus has shape (H, W) and all entries are nonnegative.

Your tasks are as follows:

1. Follow instructions and implement functions in *visualizers/saliency_map.py*, which manually computes the saliency map

2. Follow instructions and implement Saliency Map with Captum in *root/saliency_map.py*

As the final step, you should run the corresponding section in *root/main.ipynb* to generate plots for visualization.

1.2 GradCam

GradCAM (which stands for Gradient Class Activation Mapping) is a technique that tells us where a convolutional network is looking when it is making a decision on a given input image. There are three main stages to it:

1. Guided Backprop (Changing ReLU Backprop Layer, Link)

2. GradCAM (Manipulating gradients at the last convolutional layer, Link)

3. Guided GradCAM (Pointwise multiplication of above stages)

In this section, you will be implementing these three stages to recreate the full GradCAM pipeline. Your tasks are as follows:

1. Follow instructions and implement functions in *visualizers/gradcam.py*, which manually computes guided backprop and GradCam

2. Follow instructions and implement GradCam with Captum in *root/gradcam.py*

As the final step, you should run the corresponding section in *root/main.ipynb* to generate plots for visualization.

1.3 Fooling Image

We can also use the similar concept of image gradients to study the stability of the network. Consider a state-of-the-art deep neural network that generalizes well on an object recognition task. We expect such network to be robust to small perturbations of its input, because small perturbation cannot change the object category of an image. However, [2] find that applying an imperceptible non-random perturbation to a test image, it is possible to arbitrarily change the network's prediction.

[2] Szegedy et al, "Intriguing properties of neural networks", ICLR 2014

Given an image and a target class, we can perform gradient ascent over the image to maximize the target class, stopping when the network classifies the image as the target class. We term the so perturbed examples "adversarial examples".

Read the paper, and then implement the following function to generate fooling images. Your tasks are as follows:

1. Follow instructions and implement functions in *visualizers/fooling_image.py*, which manually computes the fooling image

As the final step, you should run the corresponding section in *root/main.ipynb* to generate fooling images.

1.4 Class Visualization

By starting with a random noise image and performing gradient ascent on a target class, we can generate an image that the network will recognize as the target class. This idea was first presented in [1]; [3] extended this idea by suggesting several regularization techniques that can improve the quality of the generated image.

Concretely, let I be an image and let y be a target class. Let $s_y(I)$ be the score that a convolutional network assigns to the image I for class y; note that these are raw unnormalized scores, not class probabilities. We wish to generate an image I^* that achieves a high score for the class y by solving the problem

$$I^* = \arg\max_I s_y(I) - R(I)$$

where R is a (possibly implicit) regularizer (note the sign of $R(I)$ in the argmax: we want to minimize this regularization term). We can solve this optimization problem using gradient ascent, computing gradients with respect to the generated image. We will use (explicit) L2 regularization of the form

$$R(I) = \lambda \|I\|_2^2$$

and implicit regularization as suggested by [3] by periodically blurring the generated image. We can solve this problem using gradient ascent on the generated image.

[1] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014(https://arxiv.org/abs/1312.6034)

[3] Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML 2015 Deep Learning Workshop

Your tasks are as follows:

1. Follow instructions and implement functions in *visualizers/class_visualization.py*, which manually computes the class visualization

As the final step, you should run the corresponding section in *root/main.ipynb* to generate the visualizations.

2 Style Transfer

Another task closely related to image gradient is style transfer. This has become a cool application in deep learning with computer vision. In this section we will study and implement the style transfer technique from:

"Image Style Transfer Using Convolutional Neural Networks" (Gatys et al., CVPR 2015).

The general idea is to take two images (a content image and a style image), and produce a new image that reflects the content of one but the artistic "style" of the other. We will do this by first formulating a loss function that matches the content and style of each respective image in the feature space of a deep network, and then performing gradient descent on the pixels of the image itself.

In this assignment, we will also use SqueezeNet as our feature extractor which can easily work on a CPU machine. Similarly, if computational resources are not any problem for you, you are encouraged to try a larger network, which may give you benefits in the visual output in this homework.

2.1 Content Loss

We can generate an image that reflects the content of one image and the style of another by incorporating both in our loss function. We want to penalize deviations from the content of the content image and deviations from the style of the style image. We can then use this hybrid loss function to perform gradient descent **not on the parameters** of the model, but instead **on the pixel values** of our original image.

Let's first write the content loss function. Content loss measures how much the feature map of the generated image differs from the feature map of the source image. We only care about the content representation of one layer of the network (say, layer ℓ), that has feature maps $A^\ell \in \mathbb{R}^{1\times C_\ell \times H_\ell \times W_\ell}$. C_ℓ is the number of channels in layer ℓ, H_ℓ and W_ℓ are the height and width. We will work with reshaped versions of these feature maps that combine all spatial positions into one dimension. Let $F^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ be the feature map for the current image and $P^\ell \in \mathbb{R}^{N_\ell \times M_\ell}$ be the feature map for the content source image where $M_\ell = H_\ell \times W_\ell$ is the number of elements in each feature map. Each row of F^ℓ or P^ℓ represents the vectorized activations of a particular filter, convolved over all positions of the image. Finally, let w_c be the weight of the content loss term in the loss function.

Then the content loss is given by:
$L_c = w_c \times \sum_{i,j}(F^\ell_{ij} - P^\ell_{ij})^2$

1. Implement Content Loss in *style_modules/content_loss.py*

You can check your implementation by running the tests in the notebook *main.ipynb* in the root directory.

2.2 Style Loss

Now we can tackle the style loss. For a given layer ℓ, the style loss is defined as follows:

First, compute the Gram matrix G which represents the correlations between the responses of each filter, where F is as above. The Gram matrix is an approximation to the covariance matrix – we want the activation statistics of our generated image to match the activation statistics of our style image, and matching the (approximate) covariance is one way to do that. There are a variety of ways you could do this, but the Gram matrix is nice because it's easy to compute and in practice shows good results.

Given a feature map F^ℓ of shape $(1, C_\ell, M_\ell)$, the Gram matrix has shape $(1, C_\ell, C_\ell)$ and its elements are given by:

$$G^\ell_{ij} = \sum_k F^\ell_{ik} F^\ell_{jk}$$

Assuming G^ℓ is the Gram matrix from the feature map of the current image, A^ℓ is the Gram Matrix from the feature map of the source style image, and w_ℓ a scalar weight term, then the style loss for the layer ℓ is simply the weighted Euclidean distance between the two Gram matrices:

$$L^\ell_s = w_\ell \sum_{i,j} \left(G^\ell_{ij} - A^\ell_{ij}\right)^2$$

In practice we usually compute the style loss at a set of layers $\mathcal{L}$ rather than just a single layer ℓ; then the total style loss is the sum of style losses at each layer:

$$L_s = \sum_{\ell \in \mathcal{L}} L^\ell_s$$

1. Implement Style Loss in *style_modules/style_loss.py*

You can check your implementation by running the tests in the notebook *main.ipynb* in the root directory.

2.3 Total Variation Loss

It turns out that it's helpful to also encourage smoothness in the image. We can do this by adding another term to our loss that penalizes wiggles or **total

variation** in the pixel values. This concept is widely used in many computer vision task as a regularization term.

You can compute the "total variation" as the sum of the squares of differences in the pixel values for all pairs of pixels that are next to each other (horizontally or vertically). Here we sum the total-variation regualarization for each of the 3 input channels (RGB), and weight the total summed loss by the total variation weight, w_t:

$L_{tv} = w_t \times \sum_{c=1}^{3} \left(\sum_{i=1}^{H} \sum_{j=1}^{W-1} (x_{i,j+1,c} - x_{i,j,c})^2 + \sum_{i=1}^{H-1} \sum_{j=1}^{W} (x_{i+1,j,c} - x_{i,j,c})^2 \right)$

You may not see this loss function in this particular reference paper, but you should be able to implement it based on this equation. In the next cell, fill in the definition for the TV loss term.

You need to provide an efficient vectorized implementation to receive the full credit, your implementation should not have any loops. Otherwise, penalities will be given according to the actual implementation.

1. Implement Style Loss in *style_modules/tv_loss.py*

You can check your implementation by running the tests in the notebook *main.ipynb* in the root directory.

2.4 Style Transfer

You have implemented all the loss functions in the paper. Now we're ready to string it all together. Please read the entire function: figure out what are all the parameters, inputs, solvers, etc. **The update rule in function style_transfer of style_utils.py is held out for you to finish.**

As the final step, you should run the corresponding section in *root/main.ipynb* to generate stylized images.

3 Sample Outputs

We provide some sample outputs for your reference to verify the correctness of your code:

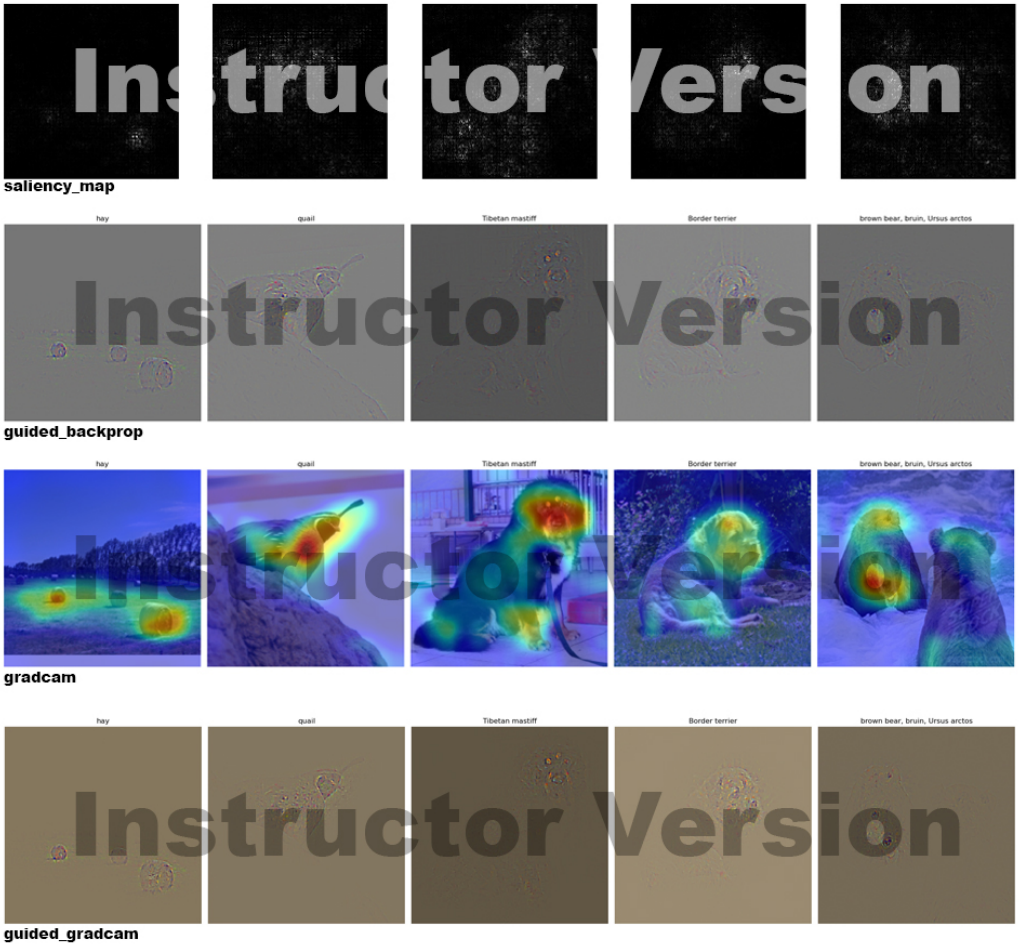

Figure 1: Example Outputs Part 1 in the following order from top to bottom - Original images, Saliency maps, Guided backprop, Gradcam and Guided Gradcam

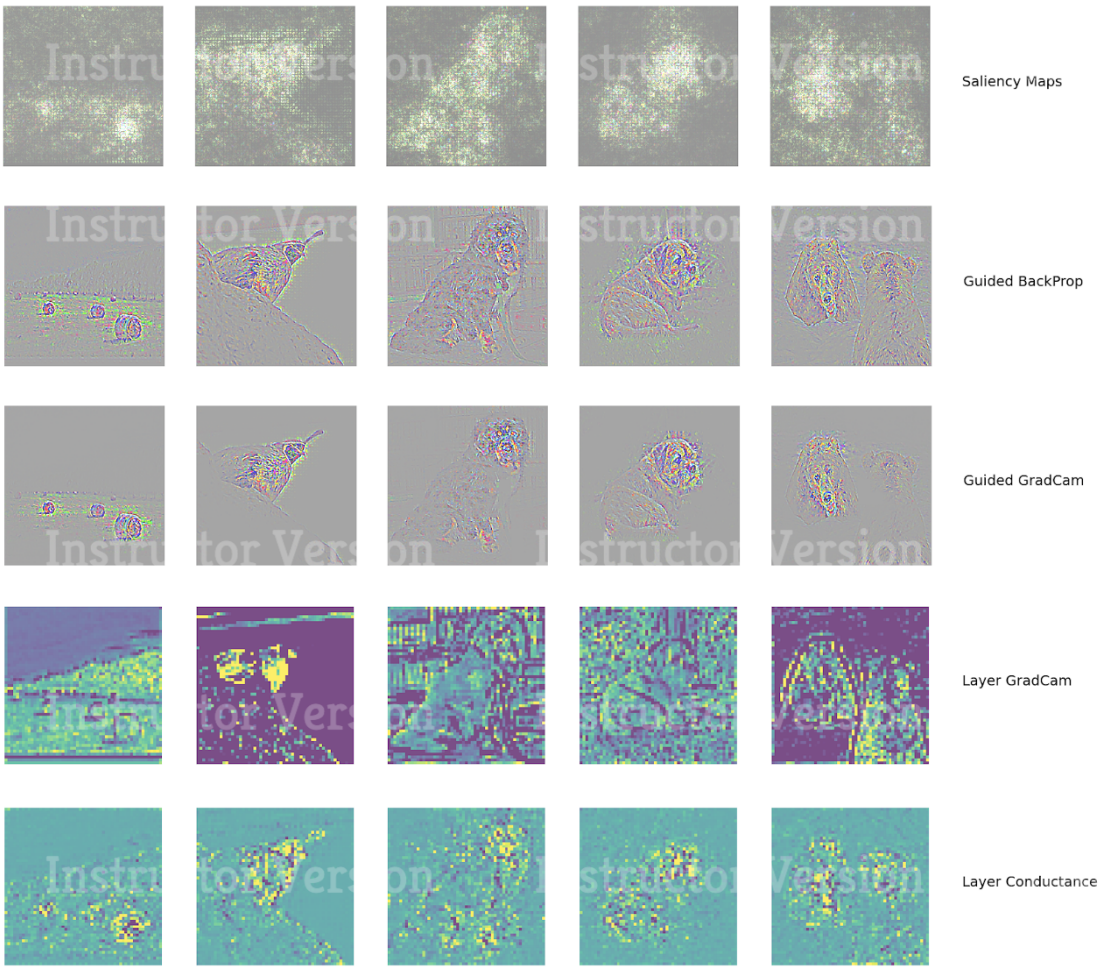

Figure 2: Example Outputs Part 2

Figure 3: Example Outputs Part 3

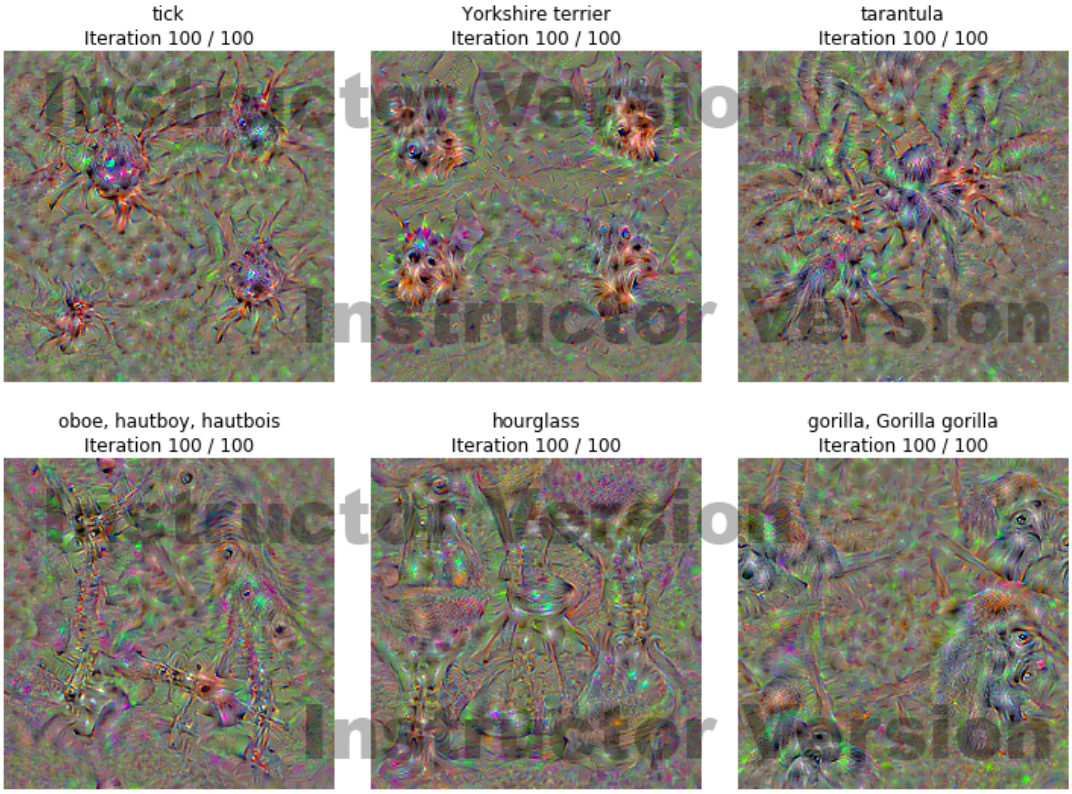

Figure 4: Class visualization

4 Deliverables

You will need to run the entire *main.ipynb* notebook, save it as a PDF, and submit it to the HW3 Report section on Gradescope. In order to convert the notebook into a PDF, you may have to download the notebook locally.

Run the script *collect_submission.py* to generate a zip folder with all the required code files. Upload the resulting zip folder to Gradescope under HW3 Code section. Only Style Transfer will be graded by the autograder.