Visualizing the Inner Workings of a CNN

Max Schneider

Problem Description

Convolutional networks are truly awe-inspiring. They can perform such a wide variety of impressive tasks, and all it takes to train them is showing them a large dataset of images, calculating the error of their guesses, and repeatedly tweaking their parameters to make the error smaller. The algorithm is so simple and elegant, yet so powerful at the same time.

The inner workings of these networks can easily go unappreciated and even unacknowledged by developers, however. Popular deep learning frameworks like TensorFlow and PyTorch make it so easy to construct a model and perform automatic differentiation that we tend to overlook the technical details of what is really going on behind the scenes. It is sometimes almost treated like some sort of all-knowing black box that you just throw your problem at and it magically learns to solve. All the developer needs to do is design the model, feed it the dataset, and it just “learns the pattern.”

But what is it that the model is actually “learning?” Can we somehow look under the hood and come up with a way to visualize how these networks actually work? The answer is: yes, we absolutely can and I will show for my project exactly how to do so.

To be more specific, I will show how to design and implement an algorithm in PyTorch that takes a pre-trained network and generates images designed to activate certain parts of the network, in order to gain a better understanding of what types of visual and geometric patterns each part of the network looks for and pays attention to.

Approach

In order to be able to explain my algorithm, it is first necessary to provide some context about the organization of convolutional neural networks (CNNs henceforth).

A CNN works by taking an input image and repeatedly parsing it to extract increasingly abstract feature maps. It is organized in layers, with most layers performing a convolution operation and some other layers in between to downsample/ condense the feature maps.

A convolution operation works by taking a “sliding window” of a fixed size and moving it across the image by some fixed step size. For each step, the window (called a “kernel” in machine learning) is “convolved” with its view of the image at that position, in a mathematical operation similar to taking a dot product. Depending on the kernel’s parameters, the kernel can serve a multitude of purposes. A convolution on the input image can detect basic primitives, like lines, edges, and corners. A good example of this is Scharr filters and Sobel filters, which are used for edge detection and computing color gradients across an image. (We will use Scharr filters later on when constructing a loss function for the optimizer.) The result of the convolution is a 2-dimensional feature map that holds the result of each convolution. When there are multiple kernels in one layer (almost always the case), a resulting 3-dimensional feature map is produced, with one channel for each kernel.

Although one single convolution operation can only detect basic patterns, applying a convolution on the information stored in the resulting feature map provides a second level of abstraction and allows for more complex geometric patterns to be detected. The more convolution layers that are stacked on top of each other, the more complex the features that the CNN can learn to recognize. With enough layers of abstraction, a CNN can perform complex tasks like image classification and face recognition, performing almost on par with the human visual system.

This way of viewing a CNN suggests a simple approach for my problem. Since each kernel in the network detects a different type of feature, we would simply need to generate an image that maximizes the activation of the kernel we are interested in if we want to visualize what kind of feature that particular kernel looks for.

Instead of training a network from scratch, we will use the pretrained GoogLeNet model, for a few reasons:

1) Training my own network takes a lot of time and resources.
2) GoogLeNet is a well-renowned model and I doubt I could make a viable competitor to it within the timeframe of this project even if I wanted to.
3) It is a very deep network with lots of different layers and kernels in each layer to investigate.

Here is a chart showing the architecture of the model:

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Although I would love to generate an image for every kernel of every layer, for the sake of time we will only focus on the first three max pool layers. I hypothesized that the deeper the kernel is in the network, the more complex and abstract the resulting image generated will be.

The first step in writing the optimization program was to design a loss function. Suppose we want to maximize the activation of kernel K in layer L. What we want to do is:

1) Run a forward pass up until layer L + 1 (we don't need to run a full forward pass and doing so is a waste of resources).
2) Return the K-th channel of layer L + 1 (this is the channel that stores the activation of the K-th kernel on each sliding window).
3) Compute the mean of this channel and set the loss function to the negative of the mean.

There are a lot of details about the implementation of this algorithm that I have left out for brevity, but all my code is available in my project submission if more details are desired.

Anyways, once I had written and tested this element of the algorithm, there were a handful of issues that I noticed:

1) The learning rate needed to generate an adequate image varied wildly by orders of magnitude from layer to layer and even kernel to kernel. This made the program very impractical to use since it required fine-tuning the learning rate for each new kernel.
2) The images were very wild and chaotic looking with sharp, drastic changes in color. (This is unsurprising considering that the network is trained to respond to edges and lines.) I would like to make the images more visually appealing.
3) The patterns being produced were on a rather small scale and required zooming in very close on the image, at which point they were too blurry to see well.

Below, I detail how I devised a solution for each of these three issues:

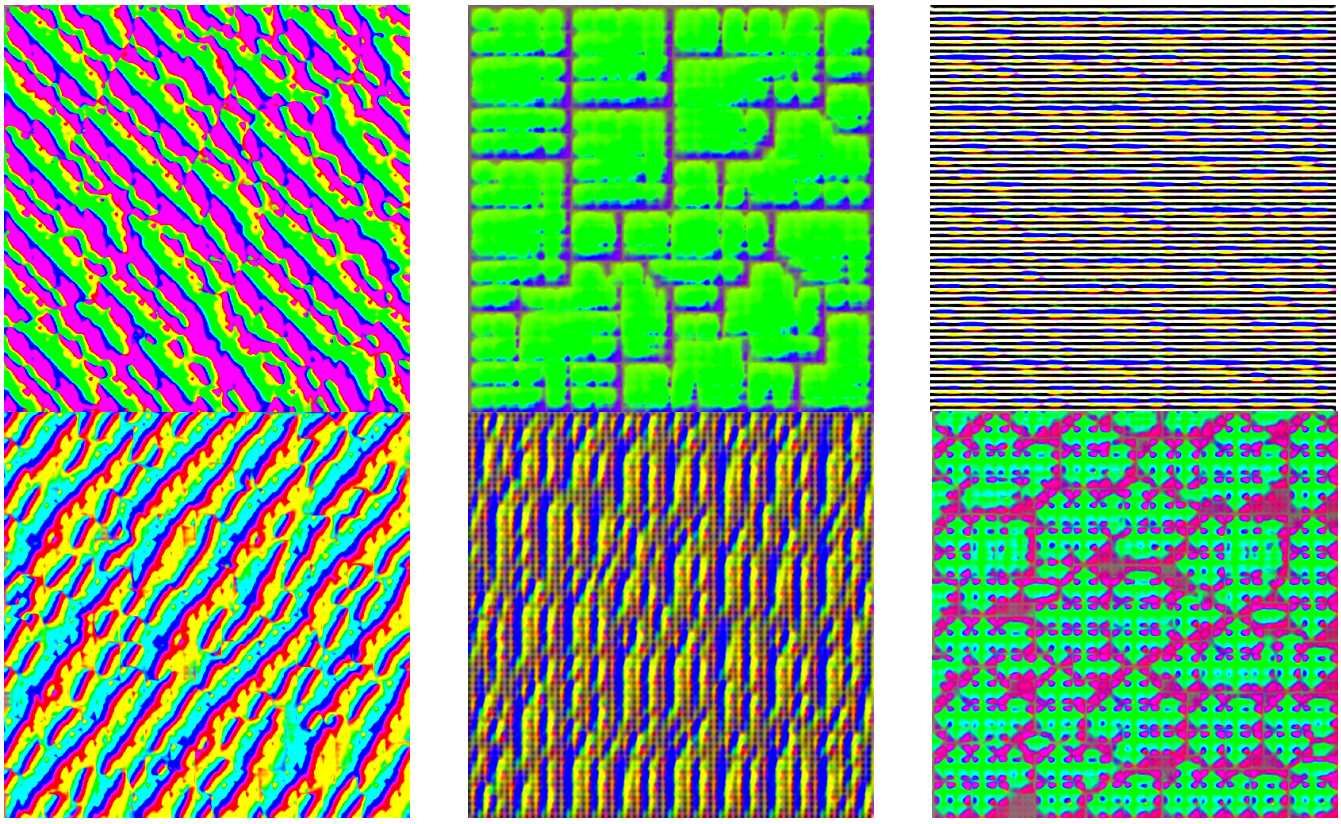
1) Instead of using PyTorch's provided optimizer classes, I wrote the gradient descent step myself in such a way that the magnitude of the gradient is standardized. Specifically, I compute *new_grad = image.grad / (image.grad.abs().max() + epsilon).* Then, I compute a fairly standard gradient descent step with this modified gradient tensor. This way, no matter how minor or extreme the gradient is, the amount by which the image changes each step is standardized and much more predictable across different kernels and layers.
2) I added two penalties in the loss function: one for high color values and another for sharp edges. The former is a trivial computation, and the latter was done using a convolution over the original image with two Scharr filters, one horizontal and one vertical. I added multipliers to each of these penalties so that the user can choose how much they should affect the resulting image via the command line interface.
3) I run the algorithm multiple times, each time scaling up the image to a new "octave." This way, there are multiple levels of detail at which the patterns can be viewed. This also gives the final image an aesthetic fractal-like appearance.

These are all the most relevant details of my implementation. As mentioned before, any further details can be deduced from looking at my code. There's not too much code and most of it is fairly straightforward.

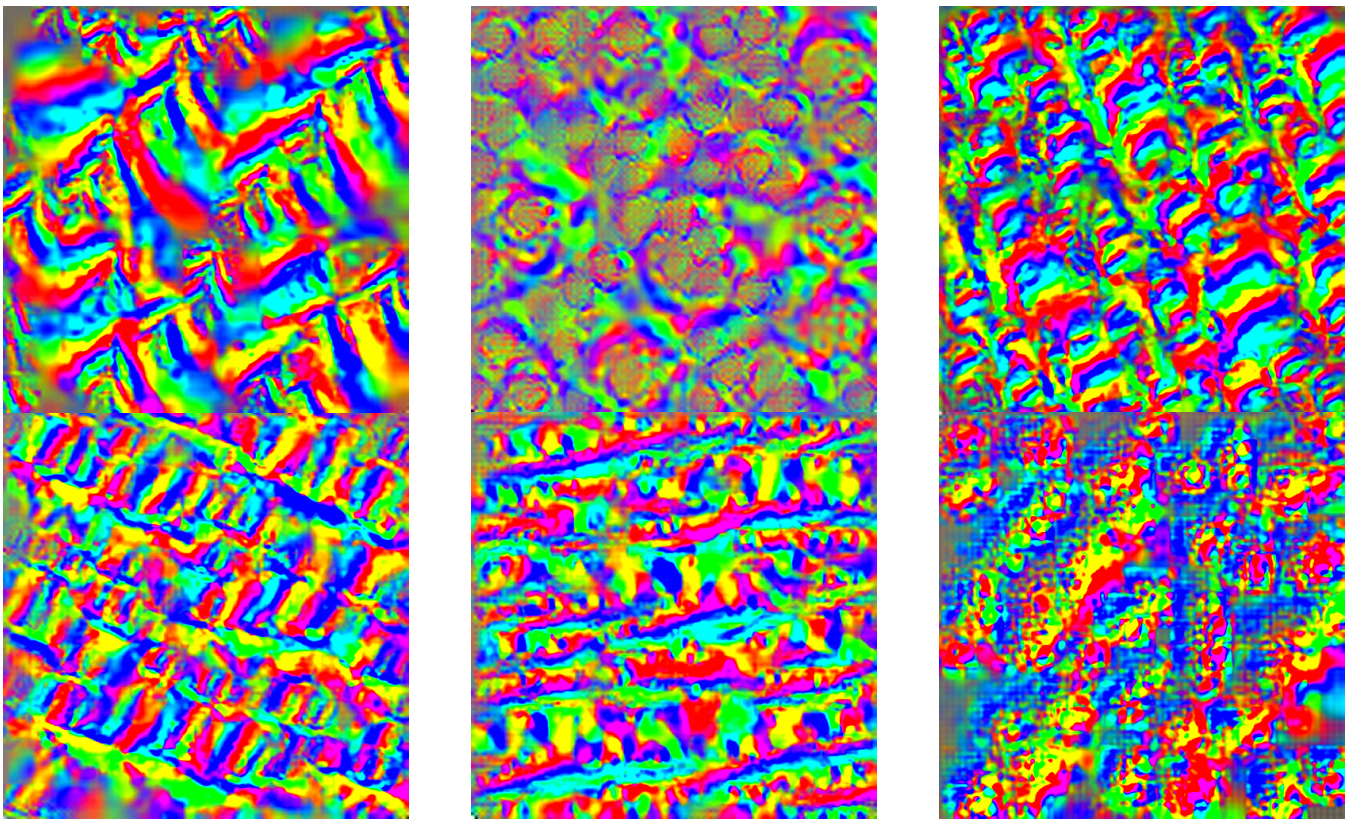
Results

I generated a batch file to run my program on every single kernel in the first three max pool layers. I will show a few from each of these layers. Please note that there is nothing particularly special about the ones that I chose, and my choices were mainly arbitrary and based on what I found to be visually appealing.

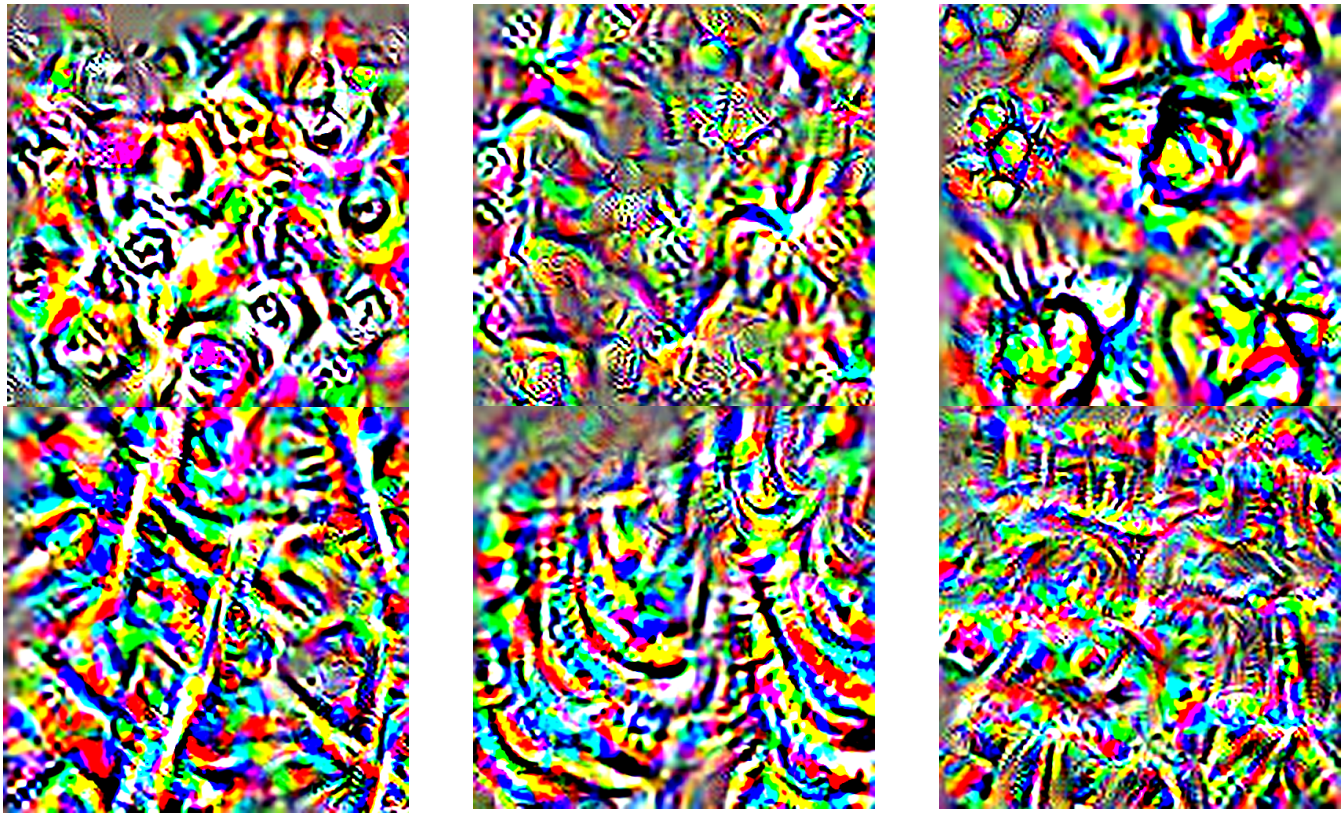
Max pool layer #1:



Max pool layer #2:



Max pool layer #3:

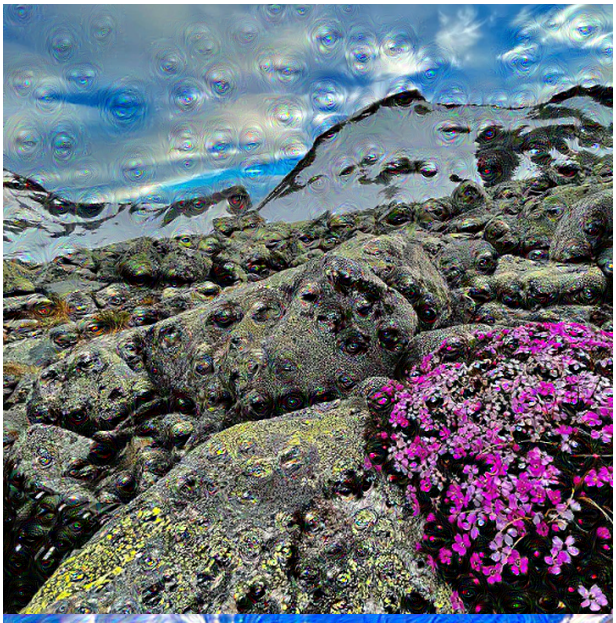


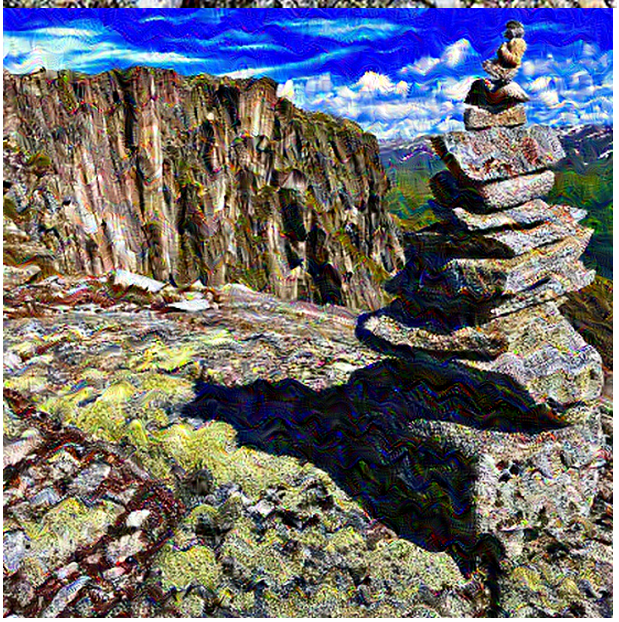
These images support my initial prediction that the network detects more complex and abstract features in deeper layers of the network.

In order to replicate these images, simply run the following commands:

- For the K-th kernel of the first max pool layer, run: *python image_gen.py --octaves 3 --size 128 --outscale 4.0 --layer maxpool1 --feature K*
- For the K-th kernel of the second max pool layer, run: *python image_gen.py --octaves 3 --size 128 --outscale 4.0 --layer maxpool2 --feature K*
- For the K-th kernel of the third max pool layer, run: *python image_gen.py --octaves 3 --size 128 --outscale 4.0 --sharpness 5.0 --layer maxpool3 --feature K* (I added sharpness for this layer in order to enhance the patterns being generated, as I just thought it looked better for this particular layer.)

At this point I realized that my program could be used to make some pretty interesting art. I added the ability to upload an external image as a starting point for the optimization, instead of just random noise. Here are some of the resulting pictures:





The changes in the second and third image are a bit subtle and might warrant zooming in. I set the step size pretty small so that the changes would be tasteful and not too drastic.

Reflection

I fully accomplished the goal that I set out to accomplish, and I even added extra features to my program just for fun. Machine learning is something I've always been super interested in and I'm very happy to be getting back into it.

This project has spurred lots of ideas for future projects that I will work on in my free time this summer:

- Continue to improve the aesthetic quality of my images.
- Extend my program to be able to maximize the output layer of the network (for instance, generating an image that causes the network to predict “dog”) and using this feature to create DeepDream-like images.
- Create an algorithm that can examine the activations in one particular feature map and trace backwards through the network to determine which pixels are most responsible for the activations. This might just sound like backpropagation but the computation would be different from simply computing derivatives. Such an algorithm could be useful for tasks like object boundary/ silhouette detection.

Justification

a) This project relates to the class because we have been studying machine learning and working with CNNs in particular recently. Additionally, my project is fundamentally about data processing and visualization, which I believe strongly fits the theme of this class.
b) I worked really hard on this project, easily spending 2-3x as much time on it as the previous homeworks. This was something I am genuinely passionate about and I spent a lot of time reading PyTorch docs, looking at source code in PyTorch’s library, and devising algorithms that would help me accomplish my goal. The task of cleaning up all my code and producing a neat command-line interface also took me considerable time.

The word count of everything up to the beginning of this sentence is: 8039 words. (Whoops, I guess I got a little carried away.)