CS165 Final Project

Tim Krasnoperov

March 21, 2019

1 Introduction and Motivation

With this project we propose and explore a modification to Google's DeepDream algorithm, a gradient-ascent/descent based approach to turn a neural network trained to classify images into a generative model. Neural network algorithms are largely treated as a black box, but visualization methods like DeepDream can shed light on their inner workings, allowing us a perspective on what features the neural network is actually learning to pick up on. As a generative "wrapper" around a classifier network, DeepDream produces surreal images that resemble the specific class it was asked to generate. Figure 1 shows a simple example of such a visualization, specifically one that demonstrates a useful facet of the algorithm.

Figure 1: Example of visualization from Google's Inceptionism Blog

This visualization tells us that the network is actually associating the presence of a dumbbell with the presence of an arm, which may or may not meet the engineer's requirements. The reason for this comes from how DeepDream works. DeepDream at it's core creates these results by altering an input to the neural network in a way that increases the activation of a chosen neuron (in the example, this neuron would be on the output layer in the index specified by the dumbbell class). This is done by taking the gradient of the input image with respect to that target class, moving the image some step in that gradient, and repeating the process. This alone, however, does not create output inseparable to the human eye. In most cases, we produce adversarial points, which convince the network of an object's presence, but look like random noise to the human eye. Such an example, generated in this project, is presented below in Figures 2. The image convinces the commonly cited AlexNet CNN that it is detecting an African Elephant (index 386 in the encoded target vector) with 96% certainty, yet it makes no sense to the human observer. AlexNet was downloaded pre-trained form the official PyTorch Github `https://github.com/pytorch/vision/blob/master/torchvision/models/alexnet.py`.

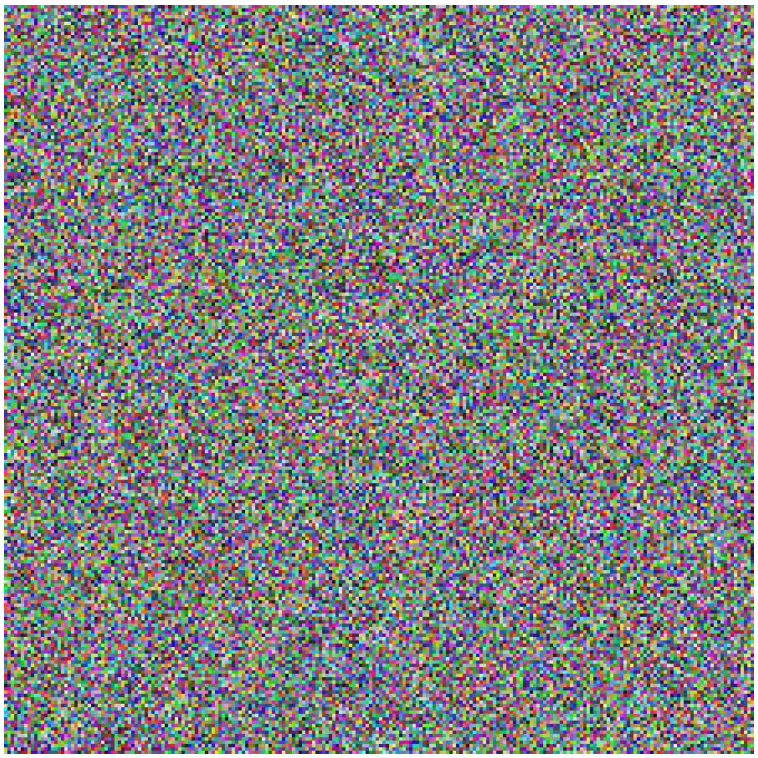

Figure 2: Project Generated Adversarial African Elephant Example

These results are a broad and studied topic on their own [1][2], but they hurt the generative process for the DeepDream algorithm. To avoid them, DeepDream uses penalty function, which direct the input image to look more natural. Generally, such functions include various blurring techniques and activation regularizers, which are finely tuned with a hyper-parameter search. This project proposes and then tests an addition to this algorithms which we hypothesized would allow for a finer control on the activations in the hidden layers. In general, we hypothesized that adding constraints on the outputs of intermediate layers would allow us some extra capabilities, such as a better convergence to the target, a perhaps better understanding of what it means for an image to be "natural", and more control on the network's perception of a generated input.

2 Problem and Algorithm

2.1 Problem Statement

Consider an input image x, a neural network M with n hidden layers, and a target final output t. We wish to add a constraint on every layer of activations (including the input and output layers, indexes by 0 and $n+1$ respectively), with a set of penalty functions $P_i \in \{P_0, P_1, ..., P_{n+1}\}$, where P_0 penalizes the input image and P_{n+1} penalizes the value of $M(x)$, the output of the network given x. We can denote the output of layer i in the model as $y_i \in \{y_1, y_2, ..., y_{n+1}\}$, where y_0 was explicitly left out as a redundancy of x. Finally, P_{n+1} will contain in it the chosen target loss function, something penalizing the distance from t to y_{n+1}. Without care for adversarial points, we could eliminate all penalty functions other than P_{n+1}, and just solve

$$\underset{x}{\operatorname{argmin}}\ P_{n+1}(x)$$

This is done with gradient ascent/descent (symmetric problem, depending on the direction of the penalty functions), using an automatic gradient library (for this project we used `Autograd` from `PyTorch`). This can, however, produce values for x which are outside of our constraints of x (we will refer to this as the domain of x). Specifically, x must be an image with a limited range of pixel values from 0 to 1. This constraint is applied with simple projection at each gradient step. If some components of x would become negative under a step iteration, we turn them to 0 instead. Similarly, if some components of x exceed 1, we turn them to 1. This leaves us with two x vectors, one where we intended to step given out gradient function, and one which was projected onto the surface of this $\mathbb{R}^{224\times224\times3}$ hyper-cube, 224 referring to the height and width of the image and 3 to the amount of color channels. After some iterations, we notice that x begins to oscillate around some point, at which we know we have achieved a local minimum and we return x (in practice, for this project, we did not monitor the convergence with every round, but rather set an iteration count which was determined by looking at average convergence rates). This x usually converges to a really good minimum, easily outperforming even all training images in terms of it's probability of a class. Such an example generated by our project is provided in Figures 2.

This algorithm can be expanded on with constraints on the input image such as blurring. Blurring of the input image can be done with a local-pixel-variation loss function, part of P_0. The new optimization then becomes

$$\underset{x}{\operatorname{argmin}}\ aP_0(x) + bP_{n+1}(x),$$

where a and b are just constants which weigh the penalty functions, and the blur loss is included in P_0. Some [3] implement this blur loss as a separate step in the descent algorithm, but this implementation detail does impact the abstract discussion. Clearly, we do not have to stop at P_0 and P_1. Perhaps we wish to have our convolution channels also be constrained, perhaps blurred. Here is the general problem we wish to solve.

$$\underset{x}{\operatorname{argmin}}\ \sum_{i=0}^{n+1} w_i P_i(x)$$

2.2 Initial Reasoning and Difficulty

This optimization is near impossible to do globally. For a lot of this project we tried to do so and failed repeatedly. The naive approach was to try to invert the network layer by layer, beginning at the output. At each actual neuron layer, we would take the pseudo-inverse solution and the kernel space, found using `numpy`'s linear algebra library, and try describe the set of possible solutions as the span of the null space vectors shifted by any pseudo-inverse solution. The idea was then to perform a change of basis to the new space and limit our optimization of the according layer's penalty function to that hyper-surface, in hopes of improving both the computation time of extra dimensionality and assuring that our answer perfectly matched the target. This became not-feasible and furthermore impossible when constraints were involved. If our input domain was some hyper-cube in $\mathbb{R}^{224\times224\times3}$, we can obviously not expect to generate every arbitrary vector in the space of y_1. Additionally, activation functions such as ReLu impose additional constraints at each layer of this network, creating a very complicated and incalculable polytope of possible outputs that is propagated down to the output layer. This shape can even lose it's polytope qualities if we encounter non-linear activation layers, destroying our ability to perform linear programming (which would regardless be computationally in-feasible, as we learned during this project). The important takeaway here is that we can not guarantee that there exists some x which $M(x)$ or y_{n+1} lands exactly on our target t. Even more generally, for any hidden layer activation vector y_i, we can not feasibly guarantee a solution x such that $M_i(x) = y_i$, where $M_i(x)$ is the ith layer output of M. Instead, we look to find *good* solutions, local minima/maxima.

2.3 Algorithm Proposition

The usual optimization of x is in fact such a method of getting these good solutions. We propose a generalization of this method that we hypothesize could converge to a better minimum. Consider for a minute

removing some amount of layers from the input size of the network, shortening it such that the inputs to the network now become the y_j layer. We now play a similar game as before, but within a deeper layer. The optimization now becomes

$$\underset{y_j}{\operatorname{argmin}} \sum_{i=j}^{n+1} w_i P_i(x)$$

We can repeat this step for another layer, one closer to the input, say k, this time optimizing to meet y_j as oppose to the target, but since we already optimized the penalty functions for the layers after y_j, we can forget about those functions and just worry about our distance from y_j. In other words, we replace $P_j(x)$ with $L(M_j(i), y_j)$, which is some distance-based loss function on y_j. Actually, this process does not have to end here. We can recursively choose another layer k, and perform the same optimization but with $L(M_j^k(y_k), y_j)$, where the new superscript notation on M_j^k signifies the models output at layer j given an input on layer k. Clearly k must be less than or equal to j.

$$\underset{y_k}{\operatorname{argmin}} \; L\big(M_j^k(y_k), y_j)\big) + \sum_{i=k}^{j-1} w_i P_i(x)$$

To clarify, y_j has already been found in the previous iteration. As we discussed in the previous section, this y_j may not be feasible to actually create, but this optimization will try to find a good answer that is both close and also considers the other penalty functions on each intermittent layer. We can repeat this process as many or as little times as we wish, but the final iteration must choose the actual x value as follows (assuming k was the last layer we wish to perform inter-layer optimization in).

$$\underset{x}{\operatorname{argmin}} \; L\big(M_j^k(y_k), y_j)\big) + \sum_{i=0}^{k-1} w_i P_i(x)$$

This algorithm finally returns this x value. And finally, if we consider this algorithm with no intermittent optimizations, we are left with the untouched original method. Hence we describe this proposition as a generalization of the canonical method.

3 Experiment Description

3.1 Question and Setup

Given this proposition, we wish to defend the algorithm's validity when compared with the canonical method. Concretely, we will explore some specific cases and attempt to make our algorithm improve on the original result, either in convergence rate or final optimization score. Here are two specific questions we will ask and attempt to answer in this project.

1. **Optimization Score.** Can this algorithm converge to a better minimum than the original? To answer this question we will define some specific target we wish to achieve and simply set all the other penalty functions to just output zero for simplicity. We can do this for multiple targets. As the control we will run the original algorithm, and as the test we will run the algorithm with various layers chosen for intermittent optimization. We will run for 100 iterations, filling in a table of the averaged results.

2. **Generation Speed.** Can this algorithm be used to expedite the generative process? Another experiment we can perform is to generate successful x inputs using an intermittent layer which we know gives a good answer to the final target. Clearly, the gradient of a given hidden layer is faster to compute than the gradient to the final output, so if we find a good hidden layer and take the gradient to it, we can avoid extra computation in generating points (both natural images and adversarial points). The

question is if these new points can satisfy the target objective well enough. For our sake, we simply need to generate a single good y_i value, where by good we mean that $M^i_{n+1}(y_i)$ is comparably close to the target as the original method. With this y_i, we then run the last step of the algorithm and generate a point x that becomes an adversary in faster time.

We will note that in all these cases, we are in the comfortable position that any example where this algorithm improves on the canonical method can prove its validity (not as something better generally, but just in those circumstances). On the other hand, not producing one of these examples does not mean this algorithm doesn't have them. In the latter case, however, we will personally consider this proposition to have failed.

3.2 Hypothesis and Implications

1. **Optimization Score.** Our hypothesis for the optimization score is that we will not be able to produce better results than the canonical method. Although we are hopeful that it may happen, we believe that an extensive hyper-parameter search is needed to find supportive examples. Regardless, this will be a meaningful experiment which will shed insight on the algorithm. In the later Future Work section, we would describe our ideal hyper-parameter search in detail.

2. **Generation Speed.** This part we hypothesize will actually work as required. There will of course be the overheard of developing some specific y_i activations vector, but we believe reusing it while still making different examples is completely feasible. The important thing to note would be the loss value we achieve with this shortened method. We will use our results from the first experiment to reason about the results from this one.

3.3 Optimization Score Results

We run the experiment for 5 total targets, each of which is a one-hot encoded class. We arbitrarily choose the classes [54, 120, 255, 386, 954] based on nothing more than our liking of those particular animals. We also choose a few special layers which we will use as out intermittent layer hyper-parameters in the algorithm. These layers were chosen such that they each had at least one convolutional layer between them, as we think these are the most interesting to separate optimizations on. We end up choosing layers [3, 6, 10, 14], and some combinations of these layers described in the summary table below. Finally, we run the algorithm 100 times for each combination of target and intermittent-layer set, taking the average final loss of x with respect to the MSE of it's output from the model and the given target. Below are the results, with each row pertaining to a specific target and each column to a specific set of intermittent layers. Note that the first column is the control column, as the empty set of intermittent layers is the original algorithm, as discussed.

	∅	[3]	[10]	[3, 6]	[6, 14]	[3, 6, 10]	[3, 10, 14]	[3, 6, 10, 14]
54	0.0004826	0.0010287	0.0010033	0.0013369	0.0011738	0.0010048	0.0010170	0.0010022
120	0.0002503	0.0010480	0.0011223	0.0013183	0.0010052	0.0010292	0.0010049	0.0010052
255	0.0002382	0.0011077	0.0010034	0.0012194	0.0010040	0.0010048	0.0010028	0.0010094
386	0.0002518	0.0010974	0.0009971	0.0011108	0.0011462	0.0010038	0.0010424	0.0010139

All across the board we see that the proposed algorithm fails significantly. In the first case it was off by a factor of over 2, and in the following cases a factor of about 4, which in the scope of possibilities could have been way worse (after all, a loss of .001 still correlates to the intended classification in the vast majority of cases). Anyway, this algorithm still fails to improve on the non-general version for the given situation, but as we will discuss in the Future Work section, this does not mean it is to completely be forgotten and dis-considered.

3.4 Generation Speed Results

Given the lackluster results of the last section, we choose the very first layer after the input as our "generative pivot". Specifically, we perform gradient ascent on a tensor of the same shape as the outputs of this layer, but for more steps with a finer step size (remember, this initial generation of our pivot y_1 needs to happen only once, so we can afford to take our time computing it). We end up with a $_1$ value with a loss of 4.188×10^{-5} using our algorithm. And now, we create these x values using the last segment of the algorithm, but reusing the same y value. With a step size of .1 and 100 steps a run for each approach, we average over 100 runs to attain our results. The original algorithm still wins the accuracy, but insignificantly, averaging a score of 0.0009977977 as compared to 0.0010066361865 by the shortened one. The time difference, however, is incredible, with the shortened one averaging under 1/3 of a second with each production and the original one taking over 28! Seems that this algorithm is a viable way to generate adversarial points quicker, given that we put in some minor overhead (keep in mind that generating this overhead is even quicker than generating a single adversary the original way). These points, however, possibly suffer from some diversity, as will be discussed in future work.

4 Future Work

This project has a lot of possible paths to take and a lot of room to grow. Specifically, with the amount of dimensionality in both the hyper-parameters of the algorithm, the definition of the problem's penalty functions, and even the actual model, this algorithm is hard to tune and reason about. In retrospect, this algorithm likely does a better job when we have highly-weighted penalties on intermittent layers. Future word would include defining some inter-layer penalty functions and attempting to make it so that the recursive method we propose gets a better score by iteratively maximizing each one. Perhaps the whole algorithm in theory is not ideal, and that a greedy optimization will most always lose out to the complete one.

As far as the generative speed goes, this algorithm clearly creates adversarial examples quicker than the original method. There stands to exist, however, an exploration of the quality and diversity of these generated points. Depending on the purpose of generating these points, perhaps as an adversary or perhaps as the engineer hoping to improve their network, this method of generation may create a more controlled yet non-diverse adversarial point due to the different generation method.

5 Conclusion

This project served a huge role in our developing understanding of neural networks. Specifically we learned a lot about the trade-off between the power of such a network and the amount of grasp we have on it's inner workings. We spent a lot of effort on attempting to invert the network and find solutions to our penalties algebraically without gradient methods just to see the obstacles grow and grow until the conclusion was clear and we gave up. Regardless, puzzling through this and then genuinely understanding for ourselves why the task is so hard ultimately grew our understanding of the networks themselves and their immense complexity. In the future, we will keep this algorithm in the back of our minds as we likely explore this visualization idea more, and perhaps find it's place in our arsenal of tools. As far as these visualization techniques go as a whole, we developed a strong respect for the difficulty of having them produce clear results, and are interested in further exploring options to give us a clearer understanding of what is commonly considered a black-box algorithm. All the code used in the project will be available to see at `https://github.com/tkrasnoperov/recursive_generation`.

6 References

[1] `https://arxiv.org/abs/1412.6572`

[2] https://arxiv.org/pdf/1412.1897v4.pdf
[3] https://arxiv.org/abs/1506.06579