Generative Editing for Adversarial Attacks

Matei Armănașu Rajanie Prabha Sean Roelofs

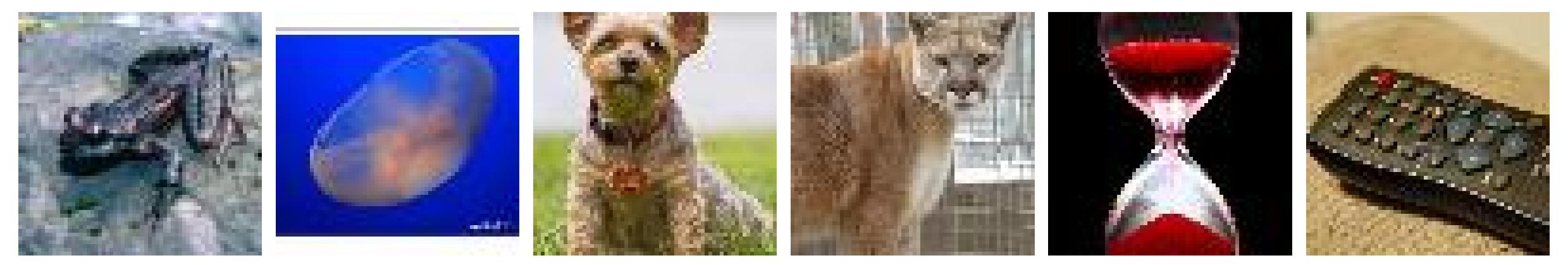
CS236 - Fall 2021

Motivation

Given the wide adoption of deep learning models, it is important to be aware on how these networks can be fooled to produce strange and potentially dangerous behaviors. An adversarial example is an instance with small, intentional feature perturbations that cause a machine learning model to make a false prediction. So far, there are many ways to generate such examples. Our goal is to explore a variant on adversarial image generation, using a generative network to produce an arbitrary quantity of edited images after training against a single classifier. We specify the fooling label which makes this a targeted attack. We compare this approach to other adversarial methods, and explore potential architectures for this adversarial generator.

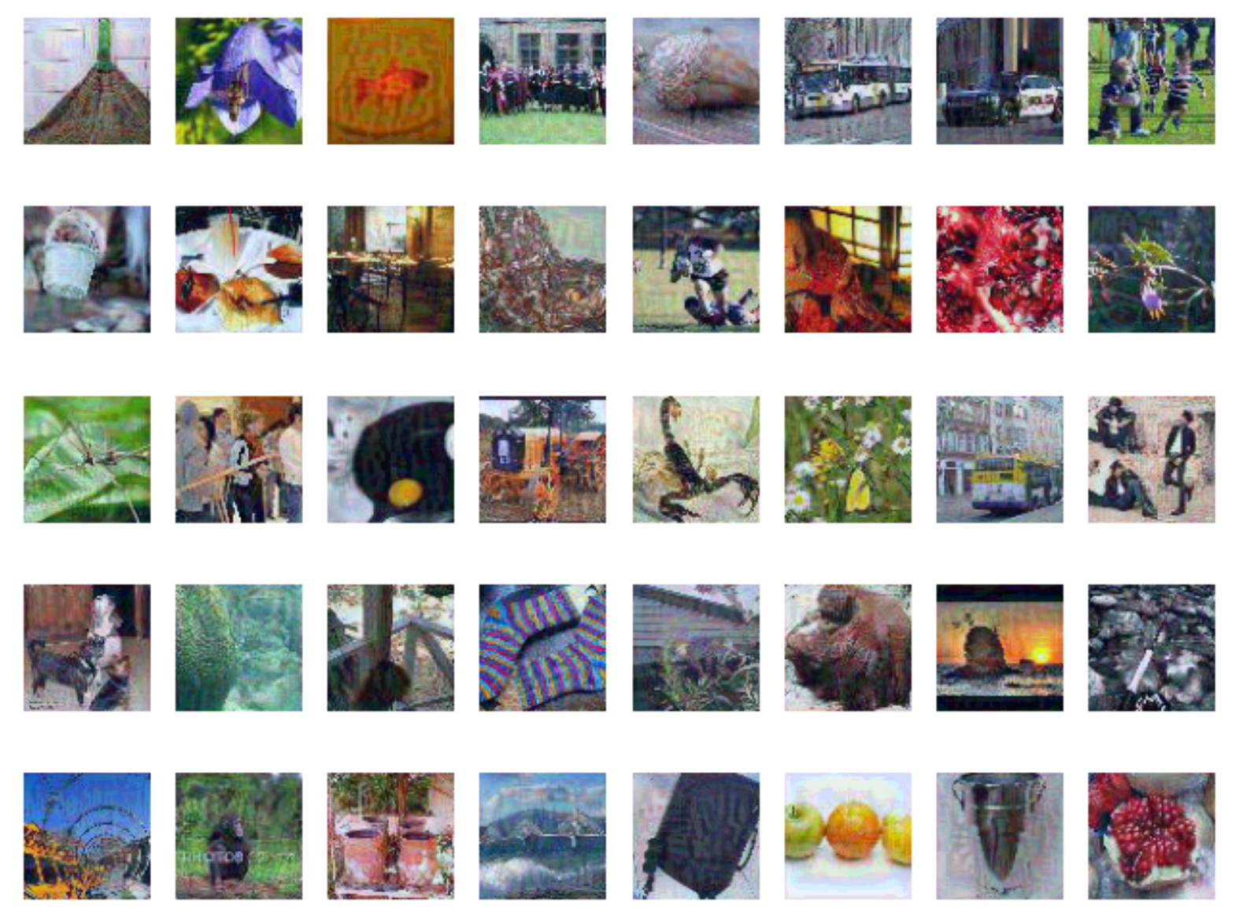
Dataset

We use the Tiny ImageNet dataset from Le and Yang [2015] for our experiments. The dataset contains 100,000 images of 200 classes (500 for each class) and each image is of $64 \times 64 \times 3$ resolution, with training, testing, and validation splits already defined across the dataset. Some samples are shown below:



Adversarial Attacks

- Images designed to trick classification networks into predicting a different class, while maintaining semantic integrity to humans.
- Relevant for understanding the weaknesses of classifiers, and can provide additional data to help guide robust design.
- Often rely on having direct access to network structure and weights (white-box) to backpropagate gradient, but black-box approaches also exist.
- Core concept is to find the shortest distance in the space of images from the original image to one which is classified differently by the network, defined by the gradient of the classification with respect to the input.
- Samples below shows example adversarial images generated by the Fast Gradient Sign Method. Target class is Praying Mantis.



Generative Network

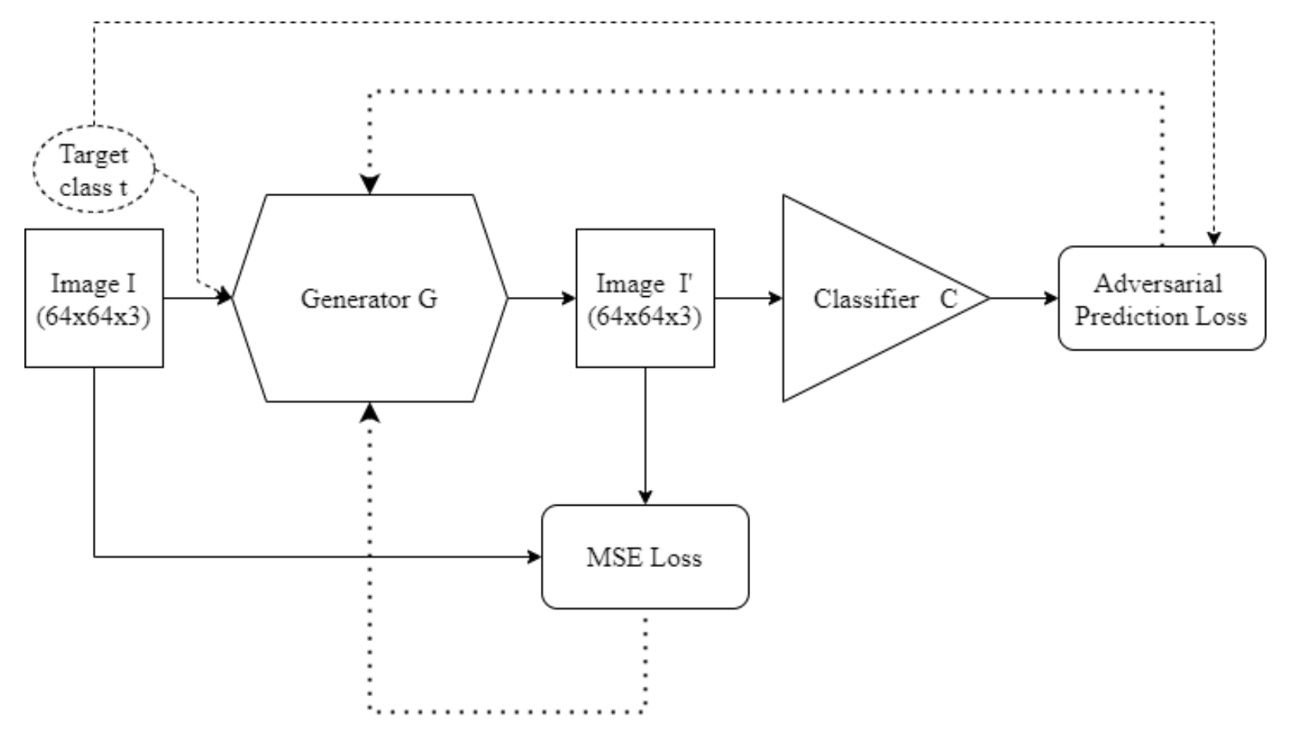


Figure 1. Our proposed training architecture. Dashed lines represent potential additions to the model for future work. Dotted lines represent loss flow.

- Generator takes in an image I as input, sized for input to a given classifier C.
- Generator outputs an image I' of the same size, perturbed such that C believes it to be from a target class t.
- Generator architecture uses a series of convolutional layers, and an optional residual mode in which the input image is added to the output to create a ResNet-style block network.
- Training process is white-box, relies on access to classifier for backpropagation.
- Loss functions (for generator $\mathcal{G}$, classifier $\mathcal{C}$, image i, target class c, tuning parameter α, and Categorical Cross-Entropy loss $CCE(\cdot,\cdot)$): $L_{MSE} = \frac{\alpha}{64\times64\times3}\|i - \mathcal{G}(i)\|_2^2$, $L_{Adv.} = CCE(c, \mathcal{C}(\mathcal{G}(i)))$

Fast Gradient Sign Baselines

- Gradient based white-box attack, focused on speed of creation. Only requires a single backpropagation step per image.
- Varying ϵ provides a trade-off between image quality and adversarial strength.
- Best empirical results maintain a SSIM above 0.9, can be used as a cutoff point for "realistic" adversarial images.
- Images from the Adversarial Attacks section were extracted from the $\epsilon = 0.2$ run.

ϵ	Metrics		
	SSIM	PSNR	Acc.
0.05	0.996	32.04	0.134
0.20	0.950	20.00	0.213
0.30	0.906	16.48	0.149
0.40	0.855	13.98	0.111
0.60	0.748	10.46	0.061

Table 1. Metrics on the results produced by FGSM for varying values of the scale parameter ϵ.

Experimental Results

All generators trained over 50 epochs on TinyImagenet. Best accuracy with 'realistic' output was 8%, significantly worse than FGSM. State-of-the-art accuracy is near 100%, but only achieved when images were obviously modified.

Experiment		Metrics		
β	LR	SSIM	PSNR	Acc.
0.0001	0.001	0.560	10.75	0.944
0.001	0.00001	0.771	17.04	0.020
0.001	0.001	0.783	14.83	0.786
0.001	0.01	0.778	14.55	0.750
0.001	0.1	0.363	9.534	0.004
0.01	0.001	0.935	22.56	0.080

Table 2. Metrics on various generator hyperparameters. Accuracy values marked in red do not outperform the natural rate at which the target class appears in the dataset.

Visualizations

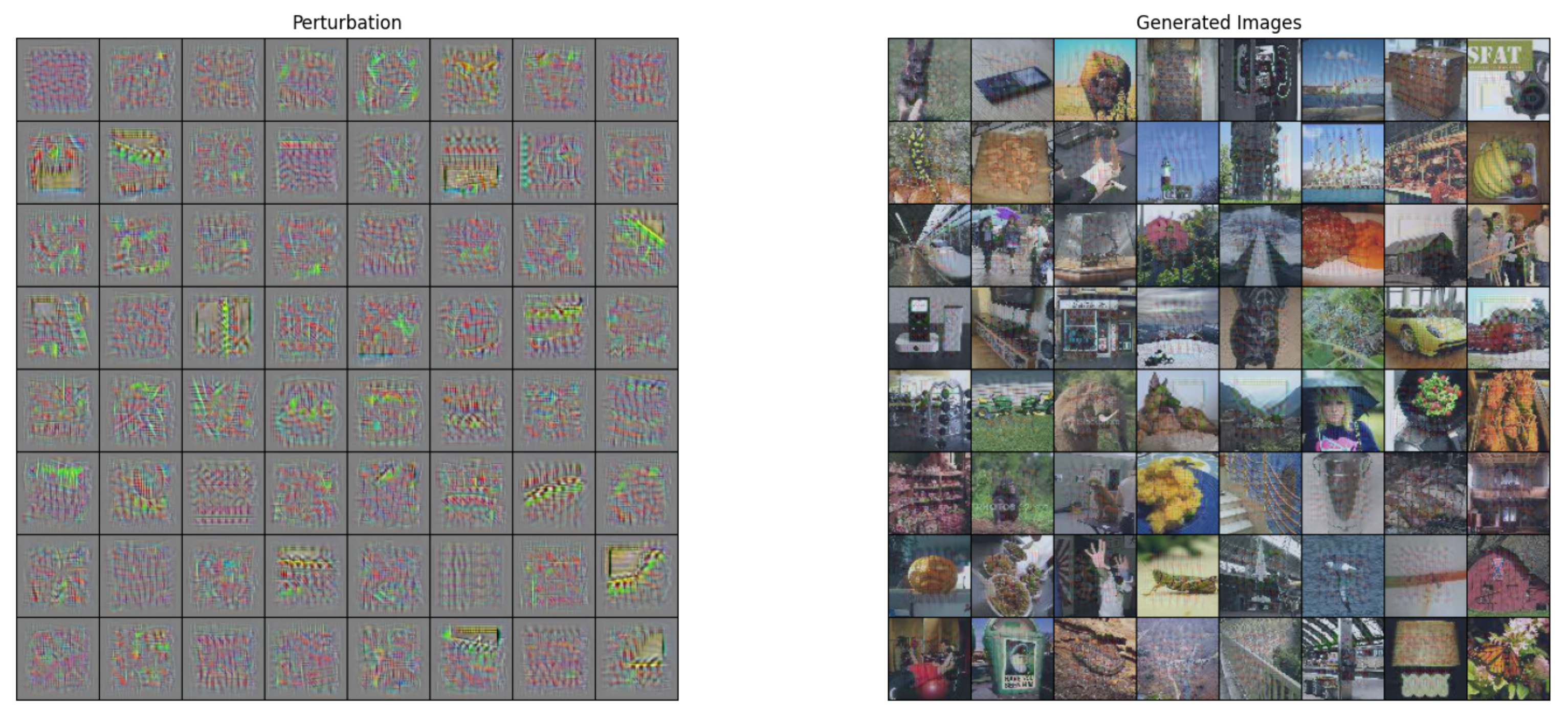


Figure 2. Left: Perturbations generated by a network trained with β= 0.01, LR = 0.001. Target class is Praying Mantis. Right: Output images with perturbations added in.

Variant Tests

- Evaluation on the training set to check for overfitting. Potential issues when both classifier and generator are trained on the same data sequentially.
- Results show little change from baselines on validation set, indicate poor performance likely from underfitting instead.
- Evaluation against EfficientNet to check for transferability, a feature observed in other papers where images made to fool one classifier work on another.
- Results show no fooling whatsoever. Possible that EfficientNet is too strong, or generator is simply too weak.

Experiment	Metrics		
	SSIM	PSNR	Acc.
Training Set	0.779	14.55	0.819
Evaluation	0.936	22.56	0.104
EfficientNet	0.781	14.55	0.004
Evaluation	0.935	22.55	0.004

Table 3. Metrics across variant evaluations to explore model strength. Tests for each variant were performed using the models from row 1 (β= .0001, LR = .001) and row 6 (β= .01, LR = .001) from Table 2 respectively.

Conclusion

Generative networks show a potential to be used for the task of adversarial image generation, but the process of training one is difficult to tune for achieving strong results. While the theory of only needing access to a classifier once and then being able to produce an arbitrary number of adversarial examples is appealing, training requires a large number of iterations and did not manage to outperform FGSM, which only requires a single backpropagation step per batch of images to be modified. Time spent resolving issues with training that stemmed from issues with the EfficientNet classifier also limited our ability to fully explore the problem, instead of checking the model performance on more well-known tasks such as image generation from random noise, redefining losses to avoid issues of saturation, and pushing hyperparameters to extremes to test loss flow.

Scan for Code: