Genetic Algorithm for Black-box Targeted Adversarial Attacks in Image Recognition

Carson Sue, Colby Meyer, Fareed Osman

School of Computing and Augmented Intelligence

Arizona State University, Tempe, AZ, USA, 85281

Abstract

The field of image recognition has made significant strides over the past few decades due to the employment of deep neural networks. State-of-the-art image recognition models, such as Google's Vision API, can distinguish between thousands of classes of images. But despite these advancements, image recognition software is still prone to failure when confronted with adversarial attacks; small perturbations to an image that can have a dramatic impact on how a model will classify it.

In this paper, we implement an algorithm for targeted adversarial attacks in a black-box setting, which is when the attacker only has query access to the image recognition model being targeted. We developed a method for creating adversarial images using the genetic algorithm, which is a black-box optimization algorithm inspired by natural selection, and tested it on examples taken from the ImageNet database.

Table of Contents

1 Background and Introduction

Adversarial attacks, broadly-speaking, fall into two categories in terms of the attacker's access to the target model; *white-box* and *black-box* adversarial attacks. In a white-box setting, the attacker has access to the loss function of the model, and can use backpropagation to find its derivative. This allows them to change an input image using gradient descent, optimizing it so that the model will classify it however the attacker wants. In black-box settings however, we do not know the loss function, and can only query the model with a given adversarial instance and see the output. Without access to the derivative of the loss function, gradient descent can't be used to directly optimize the adversarial example. This setting is the more realistic of the two, since developers who are concerned with the possibility of adversarial attacks won't make their models public.

There are a variety of different black-box optimization techniques that do not require a derivative in order to solve an optimization problem. One of the more straightforward strategies is simulation, where a new "substitute" model is trained to behave similarly to the target model, and then white-box techniques can be employed on the substitute. While this is usually effective, it comes at a high computational cost due to how long it takes to train a model that can accurately mimic the target[1]. For this reason, and considering our limited time and computing resources, we decided not to pursue this method.

Another approach to black-box optimization is using what are called *Evolutionary Strategies* (ES). These are algorithms that take inspiration from the biological processes involved in evolution, such as natural selection. Like substitution, ES can perform optimization without the need for backpropagation, but they have the additional benefit of requiring fewer computational resources and hyperparameters to implement[2]. For this project, we implement a specific algorithm from the family of ES, called the *genetic algorithm*.

The genetic algorithm is based primarily on 3 main phases[3]: selection, crossover, and mutation. After generating an initial population, the algorithm cycles through the 3 phases by selecting members of the population based on their *fitness*. The fitness values are calculated through an objective-function that measures the performance of the specimen according to whatever metrics we are trying to optimize. For our purposes, this metric would be the probability that the example will be classified with the target label.

The image recognition model we chose to attack is the VGG16 convolutional neural network model for image recognition, which includes 1000 class labels. We apply our algorithm to the pre-trained version of the model that was trained using the ImageNet database. Along with this model, the algorithm takes two images as its input; a base image from which the adversarial example should be constructed, called "original", and an image of an example of the target label, called "target". The "target" image is fed to the VGG16 model to extract the target label, and then the algorithm can be applied to optimize a noise vector. The metrics for optimization are the probability of [original + noise] being classified as the target, as well how small the noise vector is.

2 Implementation

Our implementation is mostly inspired by the work done by Chen et al.[4], who created an algorithm for perturbation-based black-box adversarial attacks that used the genetic algorithm. The paper by Chen et al. does not contain a link to their source code, but instead has high-level descriptions of their implementation along with pseudo code. We made several modifications when implementing our algorithm, mainly to deal with problems that we encountered, but also due to our constraints when it came to data and computing resources.

The APIs we used to implement our solution were Keras, a deep-learning API for Python, and NumPy, which we used to handle most of our data structures. The Pillow imaging library (PIL) was also used to handle the image data. Specific code for how to convert images to the right data structures and feed them to the VGG16 model was sourced from *PyImageSearch*[5].

2.1 Noise vectors and the initial population

The first stage of the genetic algorithm is to instantiate a starting population. Since we are attempting to optimize a noise vector, one strategy is to create the starting population from randomly generated noise. In the implementation by Chen et al., the starting population doesn't consist of randomly generated noise. Instead, they generate a number of adversarial examples from other models using white-box methods, and then use those adversarial examples as the initial population[4]. However, we decided to see if the genetic algorithm could succeed when random Gaussian noise is used to generate the starting population.

```
def rand_noise():
    noise = np.random.normal(0, sigma, (h,w))
    noise[noise > 255] = 255
    noise[noise < -255] = -255
    np.int_(noise)
    noise = np.reshape(noise, (h,w,1))
    noise = np.repeat(noise, 3, axis=2)

    return noise

rand_noises = [rand_noise() for _ in range(population_size)]
```

Here we use a distribution with a mean of 0, since noise should be able to both black-out and white-out a given pixel. For the initial population, the standard deviation *sigma* is set to be 100, but then it gradually decreases over the runtime. The values of *h* and *w* come from global variables representing the dimensions of the original image. For our implementation, the noise value being added should be the same for all three of the RGB values per pixel, so values from the Gaussian distribution are generated for each pixel and then repeated three times. We do not use overflow and underflow when adding noise to the image; if a pixel value in [original + noise] exceeds 255 or goes below 0, it "snaps" back to that bound instead of wrapping into the opposite side of the spectrum.

2.2 Fitness function

In order to implement the genetic algorithm and "evolve" our initial population, an objective-function that can evaluate each specimen's fitness is needed. As we have already mentioned, our goal is to optimize for the probability of our adversarial example being classified as the target label, as well as to minimize the size of the noise.

The first parameter (classification probability) is straightforward and can be derived by feeding the example to the VGG16 model and seeing its output.

```
def model_score(adversaries):
    """For brevity, some code is not shown. See the appendix for full source-code"""
    ...
    preds = model.predict(batch)
    preds = decode_predictions(preds, top=1000)
    probs = []
    for pred in preds:
        for _, label, prob in pred:
            if label == target_label:
                probs.append(prob)
                break
    return probs
```

VGG16 will output confidence values for each of the 1000 labels, so the value for our desired target label is fetched. Probabilities are calculated in batches for improved efficiency.

For our second optimization parameter, we want the adversary to be as close to the original image as possible. One idea was to measure the size of the noise vector itself. However, there is some inaccuracy introduced by the fact that some noise vectors will have "redundant" noise if their magnitude is too big for the pixel they get added to. To work around this, we apply the noise to the original image to create the adversary, subtract that adversary from the original, and then calculate the *mean-squared-error* (MSE) for that difference.

```
adv = apply_noise(noise)
MSE = (np.square(original - adv)).mean(axis=None)
```

The overall fitness value should therefore be a function of these two parameters (prob and MSE). The most obvious way to calculate this fitness would be to simply take `fitness = (prob - MSE)`. However, as we found out when testing our algorithm, this creates an unbalanced objective function, because it is much easier to reduce the MSE over a generation than to increase the probability. This was particularly true in the first few generations of the algorithm, as the probability scores for random Gaussian noise vectors were usually very small (usually around 5.0×10^{-5} most times we ran the algorithm). The model would therefore quickly converge to solutions that reduced the noise to nothing, and then get stuck in a local minimum without a way to improve the probability.

To rectify this, the fitness function needed to give a greater reward for improvements to the probability when its value is very small. Making the fitness proportional to the *logarithm* of the probability was what solved this problem. This made it so that the algorithm had a strong preference for raising the probability over reducing the MSE during the earlier generations. Additionally, we also experimented with applying scalar coefficients to both values to see what worked best.

The fitness function used in our final implementation is as follows:

```
fitness = math.log(prob) - (MSE / max_mse)
```

Where `max_mse` is a hyperparameter representing the maximum tolerable MSE value for an adversarial example before the algorithm should terminate. (See source code in appendix for more details)

2.3 Selection

The selection phase of the genetic algorithm typically involves selecting the most fit individuals from the population, and then having those individuals survive into the next generation[3]. Our implementation differs from this in that we don't use selection to determine survivors, but rather to determine *crossover* candidates who are used to generate a new population of offspring. The next generation is then set by taking the most fit individuals from *both* the previous generation and the population offspring. This process is explained in more detail in section 2.6.

In the implementation by Chen et al., selection is determined by applying what is known as the "roulette wheel" method, where the likelihood of selection is equal to an individual's proportion of the cumulative fitness[4]. We initially attempted to use this method for our algorithm, but found that it had poor performance when tested. This is because the differences in fitness values between members of the population were too small for there to be strong preference for the fittest individuals, making the roulette wheel method similar to random selection.

Another popular algorithm used for selection in genetic algorithms is *tournament selection*. This involves selecting a random sample from the population to compete in a *k*-way tournament, and picking the winner of the tournament based on fitness[6]. We found that this implementation led to better convergence in our algorithm, and used it in our final implementation. The size of the tournaments was set as one of the hyperparameters.

```
participants = rand.sample(population, k=size)
winner = max(participants, key=itemgetter(0))
```

2.4 Crossover

In the genetic algorithm, crossover refers to the process recombining the genetic information of different individuals to create offspring, in the hope that some of these offspring will have greater fitness than their parents[3]. Chen et al. implement crossover by creating a random boolean matrix that is evenly split between `True` and `False` values, then using this matrix to determine which set of values from each parent will transfer to their offspring[4]. Our implementation does the same, and we did not experience any issues with this method when testing the algorithm.

```
def crossover(parent1, parent2):
    child1 = parent1.copy()
    child2 = parent2.copy()
    cross_matrix = np.random.rand(h,w) < 0.5

    child1[cross_matrix] = parent2[cross_matrix]
    child2[cross_matrix] = parent1[cross_matrix]

    return child1, child2
```

(Note that the above function is somewhat different from the source code, because of how we decided to implement the mutation phase.)

2.5 Mutation

This part of the genetic algorithm refers to the process of randomly editing the "genes" of an individual, in order to maintain diversity in the population and as a form of random search across the solution space[3]. For our implementation, we decided to only have the mutation process apply to newly generated offspring, rather than allow all members of the population a chance to mutate. The final version of our source code includes a call to the mutation function from within the crossover function (see appendix). Whether mutation occurs or not depends on the value of the global variable `mutation_chance`. This value is initially set to 1.0 and then decreases over the algorithm's runtime, as we found that having a larger mutation chance during the earlier generations improved the speed of the algorithm.

In the implementation by Chen et al., mutation is performed by multiplying a small percentage of the noise values by some value between 0 and 2[4]. However, this method had poor performance in our algorithm. While we are not entirely sure why this was the case, we suspect it was likely because of the differences between our initial populations. The initial population generated with random Gaussian noise would contain, on average, values with much larger magnitudes compared to those of the white-box adversarial examples used by Chen et al. Therefore, random multiplication may have been too slow in shrinking the values of the noise vector.

To overcome this problem, we implemented our own version of the mutation function. Values of zero would be mutated by replacing them with new Gaussian noise values. Non-zero values were divided into two groups–large and medium–based on their size relative to the current value of sigma. Large values would be halved, and medium values would be set to zero. This method improved the speed of the algorithm by allowing mutations to get rid of large noise values more quickly.

```
def mutate_noise(noise):
    frequency = np.random.rand(h,w) < mutation_size

    zero_vals = (noise[:,:,0] == 0)
    large_vals = (np.absolute(noise[:,:,0]) > 2*sigma)
    middle_vals = ~zero_vals * ~large_vals

    zero_vals *= frequency
    large_vals *= frequency
    middle_vals *= frequency

    noise[zero_vals] = rand_noise()[zero_vals]
    noise[large_vals] = np.int_(noise[large_vals] * 0.5)
    noise[middle_vals] = np.array([0, 0, 0], dtype=np.int_)
```

2.6 Replacement

The final step of our implementation of the genetic algorithm is where we replace the previous generation with the next. As mentioned before, our algorithm creates new generations by selecting the fittest individuals across both the previous generation and their offspring. We do this by maintaining a min-heap for the population based on each individual's fitness. When a population of offspring is created through crossover and mutation, we perform a *pushpop* operation using the offspring. That is, we add an individual from the offspring population to our "main" population with a *push*, and then immediately remove the lowest-fitness individual from the population with a *pop*. This is repeated for all the offspring. Doing this means that if an individual offspring has worse fitness than all members of the main population, it will be removed immediately after being added.

```
[heapq.heappushpop(population, child) for child in children]
```

After generating the initial population and measuring their fitness values, we repeat the processes described in sections 2.3-2.6 until the termination conditions are met. These conditions are based on the hyperparameters `max_mse` and `min_prob`. The algorithm is successful if an example is generated with a probability greater than `min_prob` and that produces a lower MSE than `max_mse`.

3 Results and Discussion (A.I. Safety and Ethics)

To test the algorithm, we performed a targeted attack on an image of a soccer ball, with “brown bear” as the target label. The attack was performed successfully by two group members using their personal laptops to run the algorithm, taking 4 hours to generate an adversarial example in one case, and 2.5 hours in the other.

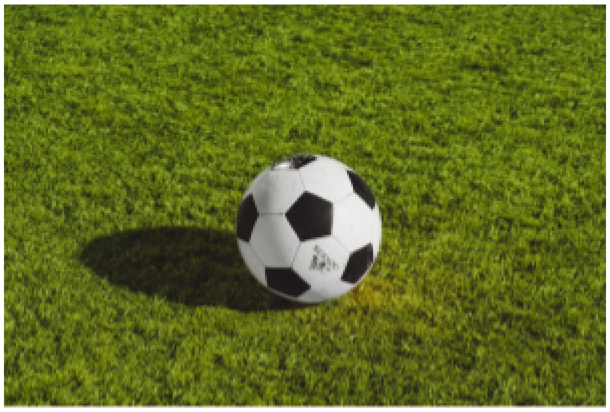


Fig. 1. Test image of a soccer ball (original.jpeg)

Fig. 2. Target image of a brown bear (target.jpeg)

In both cases, the algorithm would quickly approach confidence levels of approximately 70-80%, before slowing down as it lowered the MSE. During testing of earlier iterations of the algorithm, we had used Gaussian noise with smaller standard deviations to generate the examples, but the algorithm would find it difficult to raise the probability score to any significant value unless the program ran for multiple hours. For this reason we decided to set the standard deviation to 100 for the initial population, then gradually lower it over the runtime.

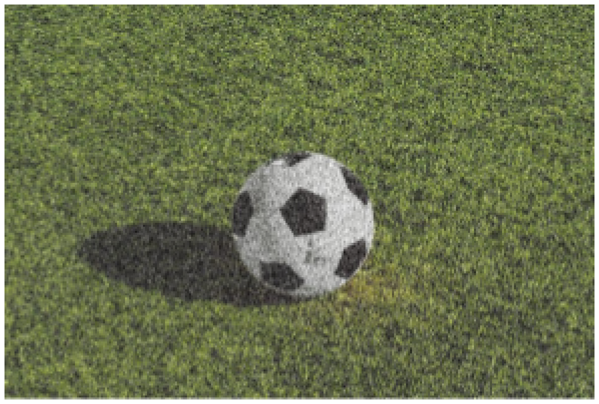


Fig. 3. Image of the soccer ball after applying one of the noise vectors from the initial population, in a test run of the algorithm

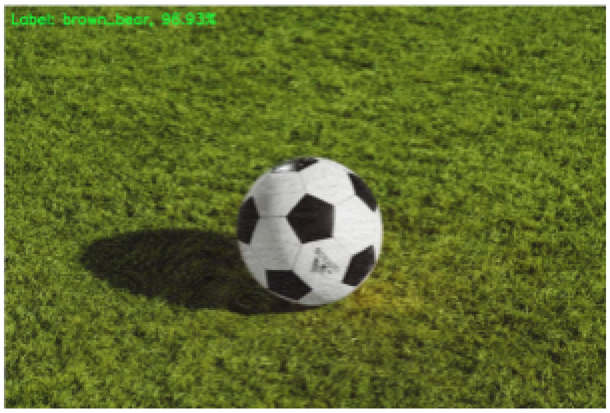
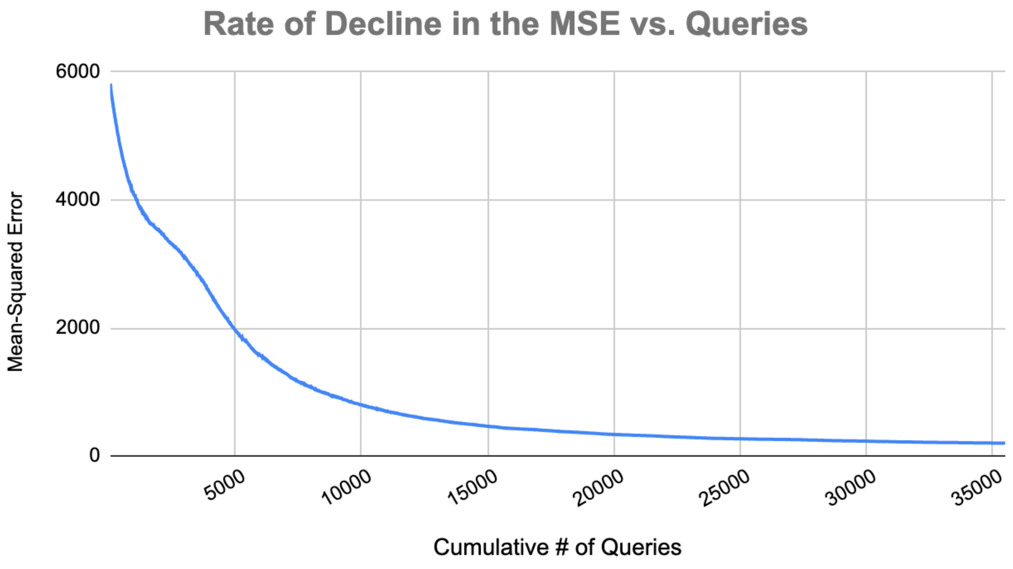


Fig. 4. Successfully generated adversarial image of the soccer ball, classified as a brown bear with 96.93% probability

Our first successful attack was the result of this strategy of having the algorithm start off with a relatively high amount of noise in the initial population. This allowed the algorithm to quickly search for solutions that contained the right noise values in the right pixels, before then moving on to shave off the unnecessary noise.

During our successful runs, the final confidence levels reached peaks of 97.24% with an MSE of 200.7 in a little over 2200 generations, after conducting a little over 35,000 queries on the model. This was based on the hyperparameters, `min_prob` and `max_mse`, which determined when the algorithm would terminate. These were set to 0.9 and 200 respectively in our final implementation.

The following are graphs of the rate of change of these two parameters over the generations for one of our successful test runs:



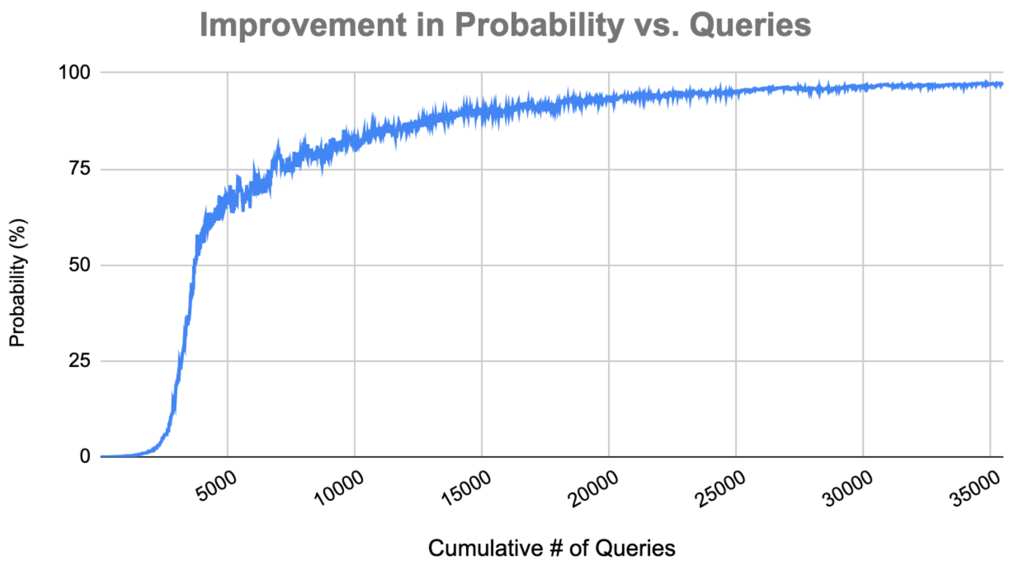


Fig. 5. Graphs of the confidence % and the MSE vs. the cumulative number of queries performed on the target model

3.1 Black-box settings and query limitations

We chose to evaluate the performance using the number of queries because it is the most consistent metric across different implementations of black-box attacks. Other metrics, such as time taken, can vary based on things like computing power which are not directly related to the efficiency of a given algorithm.

The issue of queries also ties in to the issue of how an adversarial attack could be performed in a realistic setting. While it might be easy enough for an attacker to query a target model, they will usually not be able to query it as freely as we have. A user who conducts too many queries in quick succession on a model could reasonably be suspected of having malicious intent, and thus have their access to the image recognition model blocked.

In other cases, query access to the model may come at a financial cost, such as the detection API *Clarifai*, which can require usage costs of more than \$2.40 per 1000 queries[7]. For reference, this would mean our attack on the image of the soccer ball would have cost around \$84 to perform the 35,000 queries needed. For other black-box optimization algorithms, such as the "substitution" method we mentioned before, this is a much more serious problem, as substitution can require more than a million queries to successfully implement[7]. Therefore, we can safely conclude that an attacker will always have a preference for an algorithm that can generate adversarial examples using fewer queries, all else being equal.

Another limitation to consider when discussing the potential settings of a black-box attack is how exactly the model reports its outputs. For our algorithm, we assumed that the probability value of the target label can always be retrieved, which is the case with the VGG16 model we worked with. However, some models don't report probability values for all labels, and instead will only provide outputs for the top-*k* predictions, while ignoring labels with very low probabilities[8]. Some settings are even more stringent, only providing a user with the top-*k* labels without the specific probability values, in what is referred to as a *hard-label* black-box[9].

If we wished to perform an attack with the genetic algorithm in a scenario where we only knew the top-*k* probabilities, then using random Gaussian noise in the starting population would almost certainly result in the failure of the attack, as there is very little chance that a random noise vector would create an adversary with a high enough target-label probability. A potential work-around for this issue is the strategy used by Chen et al. we previously mentioned , where white-box attacks are used to generate the starting population. This would increase the chances that an individual in the initial population would make it into the top-*k* predictions of the target model. However, this would not be a consistently successful method, as it would depend on how well the adversarial performance of the white-box models transfers to the specific black-box model being targeted.

3.2 Vulnerability to adversarial attacks

Over the course of this project, we were consistently surprised with how little prior information on the model was required by the genetic algorithm in order to perform a successful attack. As we discussed earlier, one of the main benefits of evolutionary strategies (ES) is that they do not require backpropagation of the derivative in order to bring a solution closer to the optimum. By simply performing trial-and-error, starting with completely random noise and using an appropriate objective/fitness function, it was possible for us to generate adversarial examples on a personal laptop in just a few hours, while using fewer queries than most substitution-based black-box algorithms. An attacker with more computing resources at their disposal would have had an easier time performing such an attack, while only needing to query the target model for its probability outputs.

The issue of adversarial attacks falls under the A.I. Safety pillar of "Technical Dependability", and is most directly related to the principle of "Awareness of Misuse"[10]. This is because it is the responsibility of developers to both minimize the likelihood and risks associated with the misuse of their systems. In the context of adversarial attacks on image recognition, this means not just making it more difficult for potential attackers to successfully generate adversarial examples, but also to ensure that such attacks are neither scalable nor catastrophic when they do eventually succeed. We believe that the success of the genetic algorithm in performing an adversarial attack without white-box access to the target model and with relatively little in the way of computing resources represents a significant weakness of an image recognition model.

However, despite its impressive performance, the genetic algorithm does come with its own limitations. Since the fitness function is so crucial to its performance, a poorly defined fitness function will be fatal to the algorithm, as we discovered while testing our first implementations. In some problem settings, the fitness of a solution may be ambiguous or even unknowable, such as the aforementioned *hard-label* black box problem, making the genetic algorithm nearly useless. Therefore, we believe the adoption of hard-label outputs in image recognition models would be an effective defense against such algorithms, though this would come at the cost of somewhat less useful models.

When it comes to the risks associated with successful attacks, these risks will vary wildly based on which image and model is being targeted. For example, using an adversarial image to generate a false CAPTCHA result would probably not be particularly dangerous. On the other end of the spectrum, successful attacks on a facial recognition model could have major consequences, potentially giving an attacker undeserved privileges, or providing them access to sensitive information. Developers therefore need to take these risks into consideration and balance them with the costs of adopting more restrictive measures when it comes to their image recognition models.

Given the desire many users have for clarity on how image recognition models work, some developers may be inclined to make more information on their models public in the interest of transparency (another A.I. safety principle[10]). Doing this, however, may also provide attackers even greater opportunities for conducting attacks on the model. As A.I. continues to root itself in nearly every field, researchers and developers will need to begin thinking about their work increasingly from a security perspective as well.

4 Contributions

All members of the group performed the research necessary to understand the project goal at a high level, and were able to converse and give feedback on the project topic, proposed solution, and results from the various iterations.

Fareed Osman was the primary designer of the genetic algorithm demonstrated in this model. He proposed to the group that a genetic algorithm would likely be the best method of executing a black-box adversarial attack, as well as wrote the code that can be seen throughout the paper and the appendix. Fareed also tested the code extensively, and reported back to the group with any improvements that had been made due to his testing. Additionally, Fareed wrote the Implementation section of the report, as he had the best understanding of the mechanics of our genetic algorithm.

Carson Sue assisted Fareed in testing the genetic algorithm on his own machine. Carson was able to confirm with Fareed that his results matched the results generated by Fareed's running of the script. Carson also assisted Fareed in altering the genetic algorithm in various iterations. He also contributed to the report by running extra tests of the algorithm and recording the parameter data in a file to be used for graphs and other results.

Colby Meyer attempted to find alternative methods to the genetic algorithm early in the project development when results had stagnated temporarily. However, as Fareed made more progress with the genetic algorithm, other methods of executing black-box adversarial attacks were dismissed, and the genetic algorithm was solidified as our chosen method. Colby also wrote the Results and Discussion section along with input from Fareed, and wrote the Contributions and Summary sections.

5 Summary

Artificial intelligence offers much to society in terms of technological advancement. Especially in recent years, artificial intelligence has made leaps and bounds in complex fields like medicine, data science, economics, and even business. The field of deep neural networks have been particularly influential, with their incredible capacity to learn and recognize patterns, sometimes at a higher accuracy than even humans. This has allowed them to solve problems that had previously been too complex for algorithm-based solutions. Because of this potential and how widely used artificial intelligence already is, it is extremely important to expose any security risks A.I. systems have, consider the consequences of these risks, and propose solutions to mitigate them.

In this paper, we have evaluated the capabilities of one such security risk: the vulnerabilities of neural network-based image recognition models to adversarial attacks. In theory, white-box adversarial attacks pose the most threat to A.I. models, as they can utilize backpropagation to find the derivative of a model's loss function and optimize the adversarial image using gradient descent. However, black-box adversarial attacks, as seen in this paper, can still pose a threat to artificial intelligence models, and represent a far more realistic attack setting than white-box methods. Even without knowing a model's loss function, malicious actors can still take advantage of an A.I. model by manipulating normal inputs with noise to reach a targeted output, while disguising the input as if it had not been manipulated at all.

This paper also confirmed that genetic algorithms can perform black-box attacks with high levels of accuracy. Our genetic algorithm, comprised of a *k*-way tournament algorithm for selection, a random boolean matrix to perform crossover, and our customized version of the mutation algorithm, successfully fooled the VGG16 model into classifying an image of a soccer ball as a brown bear, with confidence levels as high as 97.24%. Furthermore, our paper also suggests that adversarial attacks can be executed without high-cost computing power.

6 References

[1] S. Alarie, C. Audet, A. E. Gheribi, M. Kokkolaras, and S. Le Digabel, “Two decades of Blackbox Optimization Applications,” *EURO Journal on Computational Optimization*, vol. 9, p. 100011, 2021.

[2] A. Karpathy, T. Salimans, J. Ho, P. Chen, I. Sutskever , J. Schulman, G. Brockman, and S. Sidor, “Evolution Strategies as a Scalable Alternative to Reinforcement Learning,” *OpenAI*, Mar. 2017. https://openai.com/blog/evolution-strategies/

[3] S. Mirjalili, “Genetic algorithm,” *Studies in Computational Intelligence*, pp. 43–55, 2018.

[4] J. Chen, M. Su, S. Shen, H. Xiong, and H. Zheng, “Poba-ga: Perturbation optimized black-box adversarial attacks via genetic algorithm,” *Computers & Security*, vol. 85, pp. 89–106, Apr. 2019.

[5] A. Rosebrock, “Targeted adversarial attacks with Keras and TensorFlow,” *PyImageSearch*, 26-Oct-2020. https://pyimagesearch.com/2020/10/26/targeted-adversarial-attacks-with-keras-and-tensorflow/

[6] J. Zhong, X. Hu, J. Zhang, and M. Gu, “Comparison of performance between different selection strategies on simple genetic algorithms,” *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, Nov. 2005.

[7] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, “Black-box Adversarial Attacks with Limited Queries and Information,” *International Conference on Machine Learning (ICML)*, 2018. https://arxiv.org/pdf/1804.08598.pdf

[8] S. Bhambri, S. Muku, A. Tulasi, and A. B. Buduru, “A Survey of Black-Box Adversarial Attacks on Computer Vision Models,” Feb. 2020. https://arxiv.org/pdf/1912.01667.pdf

[9] S. Liu, J. Sun, and J. Li, “Query-efficient hard-label black-box attacks using biased sampling,” *2020 Chinese Automation Congress (CAC)*, 2020.

[10] *Ethically Aligned Design: A Vision for Prioritizing Human Well-being with Autonomous and Intelligent Systems*, The IEEE Global Initiative, New York, NY, 2017. Available: https://standards.ieee.org/wp-content/uploads/import/documents/other/ead_v2.pdf

Appendix

Full source code for our algorithm:

```
# Import packages
from PIL import Image

from keras.applications.imagenet_utils import decode_predictions
from keras.applications.imagenet_utils import preprocess_input
from keras.applications.vgg16 import VGG16
from operator import itemgetter
from itertools import count

import numpy as np
import random as rand
import math
import heapq
import cv2
import time

# Used to help break ties for heapq operations
tiebreaker = count()

# Original image we want to create an adversary from
original = Image.open("original.jpeg")
w, h = original.size
original = np.array(original, dtype=np.int_)

# Load the pre-trained model
print("[INFO] loading network...")
model = VGG16(weights="imagenet")

# Use the target image to get the target label
target = Image.open("target.jpeg")
target = target.resize((224,224))
target = np.array(target)
target = np.expand_dims(target, axis=0)
target = preprocess_input(target)

preds = model.predict(target)
preds = decode_predictions(preds, top=1000)
target_label = preds[0][0][1]
```

```
# Parameters for the genetic algorithm
population_size = 20
tour_size = 3
num_pairs = 8
queries = num_pairs * 2
mutation_size = 0.01
mutation_chance = 1.0
min_prob = 0.90
max_mse = 200
sigma = 100

# Generate a random noise vector in the dimensions of the adversarial image
def rand_noise():
    noise = np.random.normal(0, sigma, (h,w))

    noise[noise > 255] = 255
    noise[noise < -255] = -255

    np.int_(noise)

    noise = np.reshape(noise, (h,w,1))
    noise = np.repeat(noise, 3, axis=2)

    return noise

# Apply noise vector to original image
def apply_noise(noise):
    adv = noise + original

    adv[adv > 255] = 255
    adv[adv < 0] = 0

    return np.uint8(adv)

# Mutates a noise vector by halving large values,
# replacing zero values with new noise, and zeroing
# everything else.
def mutate_noise(noise):
    frequency = np.random.rand(h,w) < mutation_size

    zero_vals = (noise[:,:,0] == 0)
    large_vals = (np.absolute(noise[:,:,0]) > 2*sigma)
    middle_vals = ~zero_vals * ~large_vals

    zero_vals *= frequency
    large_vals *= frequency
    middle_vals *= frequency

    noise[zero_vals] = rand_noise()[zero_vals]
    noise[large_vals] = np.int_(noise[large_vals] * 0.5)
    noise[middle_vals] = np.array([0, 0, 0], dtype=np.int_)
```

```
# Creates two off-spring of two parents by mutating them
# and then crossing ~50% of their genes with one another
def cross_mutation(parent1, parent2):
    child1 = parent1.copy()
    child2 = parent2.copy()

    cross_matrix = np.random.rand(h,w) < 0.5

    child1[cross_matrix] = parent2[cross_matrix]
    child2[cross_matrix] = parent1[cross_matrix]

    if rand.random() < mutation_chance:
        mutate_noise(child1)
    if rand.random() < mutation_chance:
        mutate_noise(child2)

    return child1, child2

# Get the VGG16 pre-trained model's prediction of the adversarial images
def model_score(adversaries):
    batch = list(map(np.array, [img.resize((224,224)) for img in map(Image.fromarray,
adversaries)]))

    batch = np.array(batch)
    batch = preprocess_input(batch)

    preds = model.predict(batch)
    preds = decode_predictions(preds, top=1000)

    probs = []
    for pred in preds:
        for _, label, prob in pred:
            if label == target_label:
                probs.append(prob)
                break

    return probs

# Fitness function for the noises. Fitness is based on the log of
# the model's confidence that the image belongs to the target class,
# as well as the size of the noise (Mean-Squared-Error).
def fitness(noises):
    adversaries = [apply_noise(noise) for noise in noises]
    probs = model_score(adversaries)
    MSE_vals = [(np.square(original - adv)).mean(axis=None) for adv in adversaries]

    return [math.log(prob) - (mse / max_mse) for prob, mse in zip(probs, MSE_vals)]
```

```
# Selects two members of the population using random tournament selection
def selection(population, size):
    pair = []
    popu = population.copy()

    for _ in range(2):
        participants = rand.sample(popu, k=size)
        winner = max(participants, key=itemgetter(0))

        pair.append(winner[2])
        popu.remove(winner)

    return pair

rand_noises = [rand_noise() for _ in range(population_size)]
counts = [next(tiebreaker) for _ in range(population_size)]
scores = fitness(rand_noises)

population = list(zip(scores, counts, rand_noises))
heapq.heapify(population)

generation_num = 0
num_queries = population_size
top = max(population, key=itemgetter(0))
optimal = top[2]
adv = apply_noise(optimal)
prob = model_score([adv])[0]
mse = (np.square(original - adv)).mean(axis=None)
prev_fitness = fitness([optimal])

start_time = time.time()
```

```
while prob < min_prob or mse > max_mse:
    generation_num += 1
    sigma = 100*(1 - math.sqrt(prob)) + math.sqrt(prob*max_mse)
    mutation_chance = (1 - prob**2) + (prob**2)*0.2

    pairings = [selection(population, tour_size) for _ in range(num_pairs)]
    children = []
    for pair in pairings:
        children += cross_mutation(*pair)

    counts = [next(tiebreaker) for _ in range(len(children))]

    new_population = list(zip(fitness(children), counts, children))
    num_queries += queries

    [heapq.heappushpop(population, noise) for noise in new_population]

    top = max(population, key=itemgetter(0))
    optimal = top[2]
    adv = apply_noise(optimal)
    prob = model_score([adv])[0]
    mse = (np.square(original - adv)).mean(axis=None)

    print("Generation {}".format(generation_num))
    print("Label: {}, {:f}%".format(target_label, prob * 100))
    print("Mean Squared Error: {:.2f}".format(mse))
    print("Total queries made: {:d}".format(num_queries))
    print("--- {:d} minutes {:d} seconds ---".format(int((time.time() - start_time)/ 60),
int((time.time() - start_time) % 60)))
    print("##############################\n")

    Image.fromarray(adv).save("adversary.jpeg")

    if generation_num % 100 == 0:
        # Terminate if there is no improvement in fitness
        # for some number of generations
        curr_fitness = fitness([optimal])
        if curr_fitness == prev_fitness:
            print("Algorithm is too slow or has converged on suboptimal solution.")
            break
        prev_fitness = curr_fitness


image = cv2.imread("adversary.jpeg")
cv2.putText(image, "Label: {}, {:.2f}%".format(target_label, prob * 100),
    (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
cv2.imshow("Classification", image)
cv2.waitKey(0)
```