A Project Report

On

Quantum Adversarial Deep Learning

BY

Anish Kumar Kallepalli

2020A7TS0282H

Under the supervision of

Aneesh Chivukula

SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS OF

CS F491: SPECIAL PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(March 2024)

ACKNOWLEDGMENTS

I want to start by saying how grateful I am to my professor, Aneesh Chivukula, for his consistent support, wisdom, and patience during the course of this project. Their amount of information, skill, and zeal has served as a continual source of encouragement and inspiration for me. I am really appreciative of their mentoring and the chance to work with them.

I owe a great deal of gratitude to the members of the Computer Science Department staff who gave me access to the materials, tools, and equipment I needed to finish my project. This initiative is achievable thanks to their persistent efforts and commitment to their jobs.

I want to thank a few of my friends for their constructive criticism, ideas, and support during the process. Their opinions and insights have been important in helping me to develop my concepts and raise the caliber of my work.

Finally, I want to thank my family for their unwavering support, patience, and encouragement throughout my academic career. Throughout trying times, their unshakable faith in me has served as a source of courage and inspiration.

I would want to conclude by expressing my sincere gratitude to everyone who helped make this initiative a reality. I really appreciate your encouragement, support, and vital efforts.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

Certificate

This is to certify that the project report entitled "**Quantum Adversarial Deep Learning**" submitted by Mr Anish Kumar Kallepalli (ID No. 2020A7TS0282H) in partial fulfillment of the requirements of the course CS F491, Special Project Course, embodies the work done by him under my supervision and guidance.

Date: 12th March 2024 **(**Aneesh Chivukula**)**

BITS- Pilani, Hyderabad Campus

ABSTRACT

We worked on implementing an adversarial learning algorithm for supervised classification in general and Convolutional Neural Networks (CNN) in particular. The algorithm's objective is to produce small changes to the data distribution (Handwritten Digits dataset-MNIST) defined over positive and negative class labels so that the resulting data distribution is misclassified by the CNN.

The theoretical goal is to determine a manipulating change on the input data that finds learner decision boundaries where many positive labels become negative labels. Then we propose a CNN which is secure against such unforeseen changes in data. The algorithm generates adversarial manipulations by formulating a multiplayer stochastic game targeting the classification performance of the CNN. The multiplayer stochastic game is expressed in terms of multiple two-player sequential games. Each game consists of interactions between two players—an intelligent adversary and the learner CNN—such that a player's payoff function increases with interactions. Following the convergence of a sequential noncooperative Stackelberg game, each two-player game is solved for the Nash equilibrium. The Nash equilibrium finds a pair of strategies from which there is no incentive for either learner or adversary to deviate.

We then retrain the learner over all the adversarial manipulations generated by multiple players to propose a secure CNN which is robust to subsequent adversarial data manipulations. The results suggest that game theory and evolutionary algorithms are very effective in securing deep learning models against performance vulnerabilities simulated as attack scenarios from multiple adversaries.

CONTENTS

Introduction

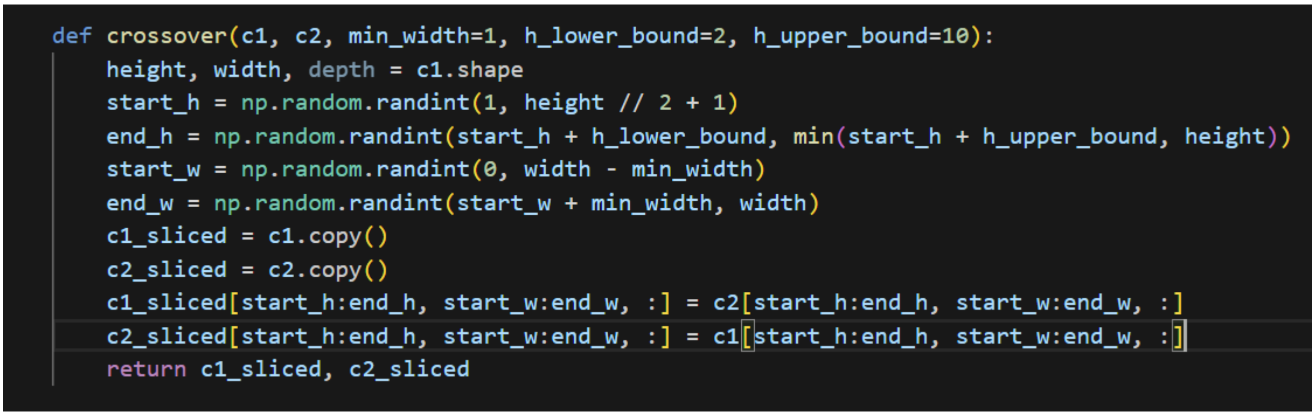
To learn mathematical patterns, machine learning methods make assumptions on the data distributions for training and testing the learning algorithm. In this report we work to implement an algorithm designed by Dr. Aneesh Chivukula. The algorithm generates a testing data distribution which is non-stationary with respect to the training data distribution. Designing robust data mining models, computing systems and machine learning algorithms for non-stationary data analytics is the goal of adversarial learning. Adversarial learning has application in areas such as spam filtering, virus detection, intrusion detection, fraud detection, biometric authentication, network protocol verification, computational advertising, recommender systems, social media web mining and performance modelling of complex systems.

Adversarial learning algorithms are specifically designed to exploit vulnerabilities in a given machine learning algorithm. These vulnerabilities are simulated by training the learning algorithm under various attack scenarios and policies. The attack scenarios are assumed to be formulated by an intelligent adversary. The optimal attack policy is formulated to solve one or many optimization problems over one or many attack scenarios. A learning algorithm designed over adversarial settings becomes robust to such vulnerabilities in the training and testing data distributions. The various adversarial learning algorithms differ in assumptions regarding the adversary's knowledge, security violation, attack strategies and attack influence.

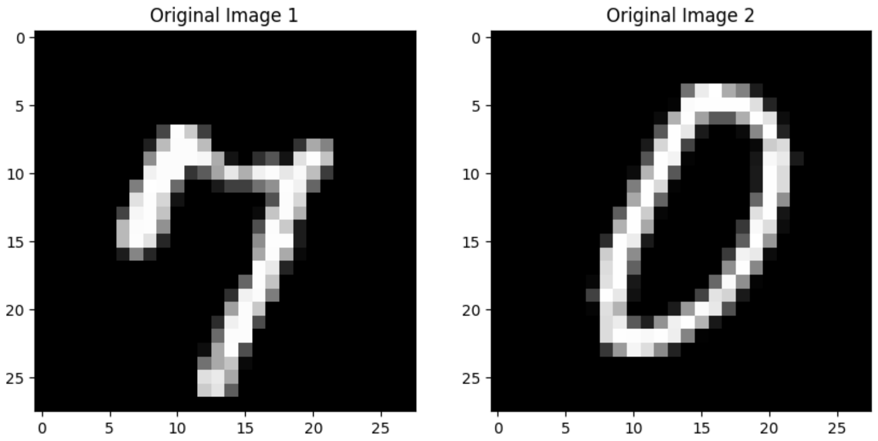
Implementation of Various Functions

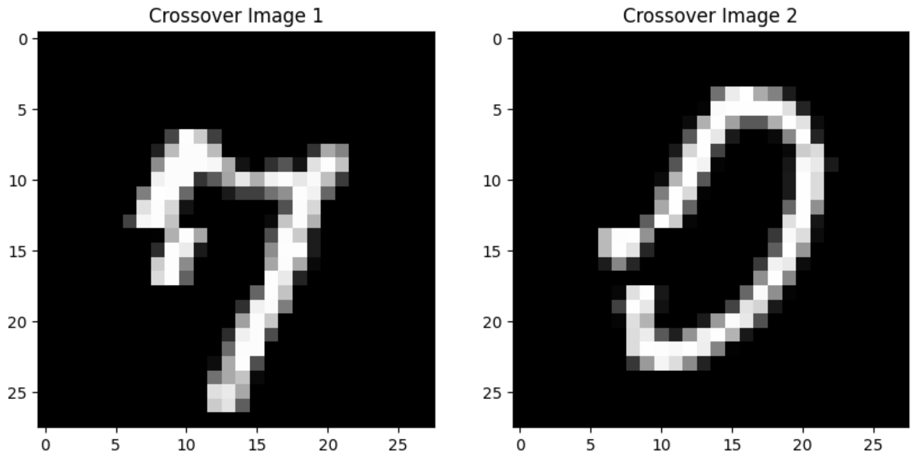
Crossover Function-

The crossover function swaps selected parts of two input arrays, c1 and c2, to create new arrays, c1_sliced and c2_sliced. This swapping mimics genetic inheritance and is often used in evolutionary algorithms to generate diverse offspring.



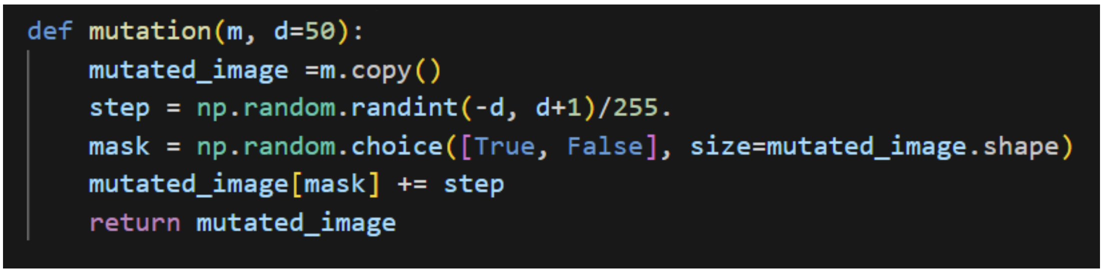
```python
def crossover(c1, c2, min_width=1, h_lower_bound=2, h_upper_bound=10):
    height, width, depth = c1.shape
    start_h = np.random.randint(1, height // 2 + 1)
    end_h = np.random.randint(start_h + h_lower_bound, min(start_h + h_upper_bound, height))
    start_w = np.random.randint(0, width - min_width)
    end_w = np.random.randint(start_w + min_width, width)
    c1_sliced = c1.copy()
    c2_sliced = c2.copy()
    c1_sliced[start_h:end_h, start_w:end_w, :] = c2[start_h:end_h, start_w:end_w, :]
    c2_sliced[start_h:end_h, start_w:end_w, :] = c1[start_h:end_h, start_w:end_w, :]
    return c1_sliced, c2_sliced
```



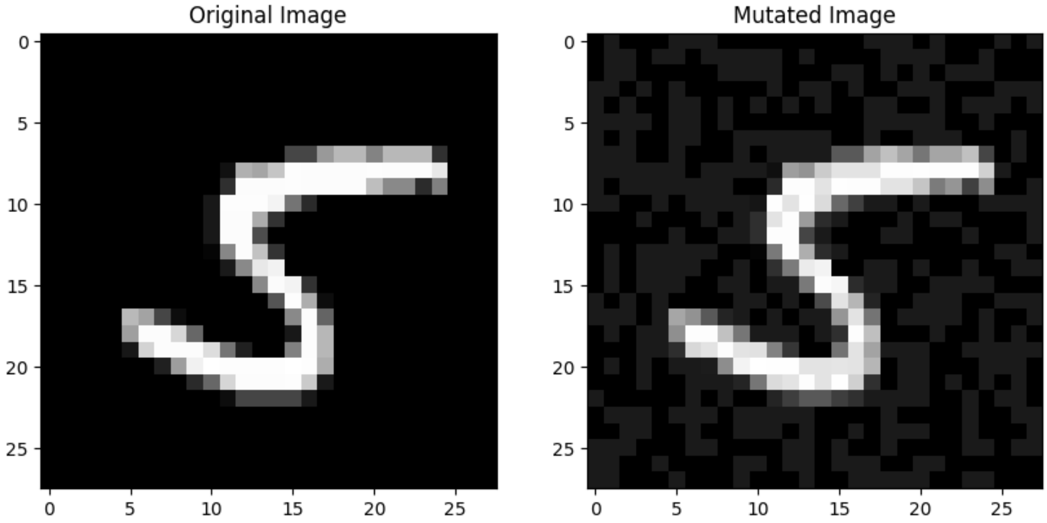


Mutation Function-

The mutation function creates a mutated version of the input image m. It randomly selects a step size within the range specified by d, then applies this step size to randomly chosen pixels in the image. This process introduces small, random changes to the image, akin to genetic mutations in biological organisms.

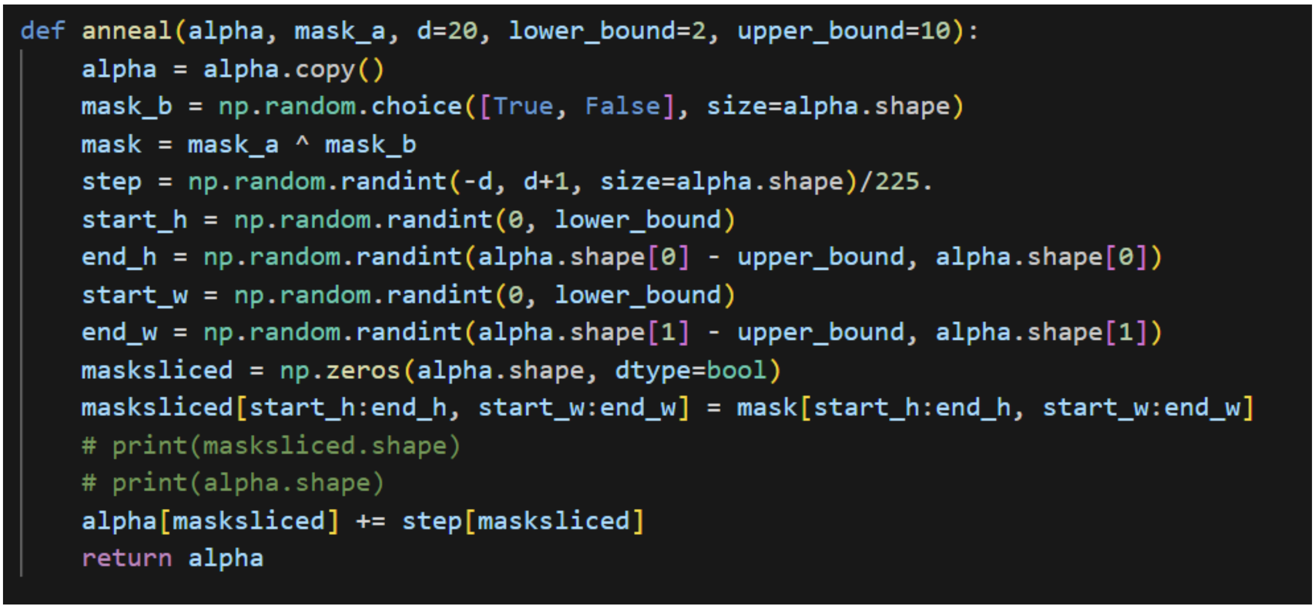


```python
def mutation(m, d=50):
    mutated_image =m.copy()
    step = np.random.randint(-d, d+1)/255.
    mask = np.random.choice([True, False], size=mutated_image.shape)
    mutated_image[mask] += step
    return mutated_image
```

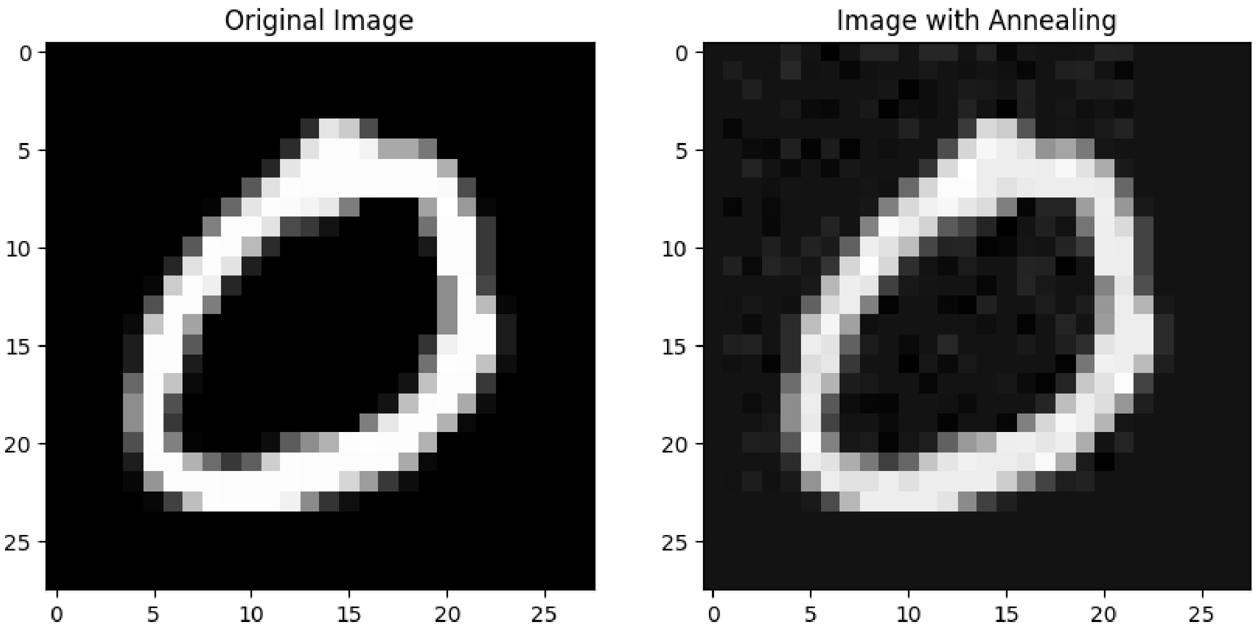


Annealing Function-

The anneal function modifies an input array alpha by applying annealing, a process inspired by simulated annealing in optimization. It first creates a random binary mask mask_b and combines it with the input mask mask_a using a logical XOR operation. Then, it selects a step size within the range specified by d and applies this step size to the elements of alpha corresponding to the combined mask. Additionally, it randomly selects a region within alpha defined by lower_bound and upper_bound and applies the step size only to the elements within this region. This process introduces controlled changes to the array, resembling the gradual cooling process in annealing, where a material's properties evolve toward an optimal state.

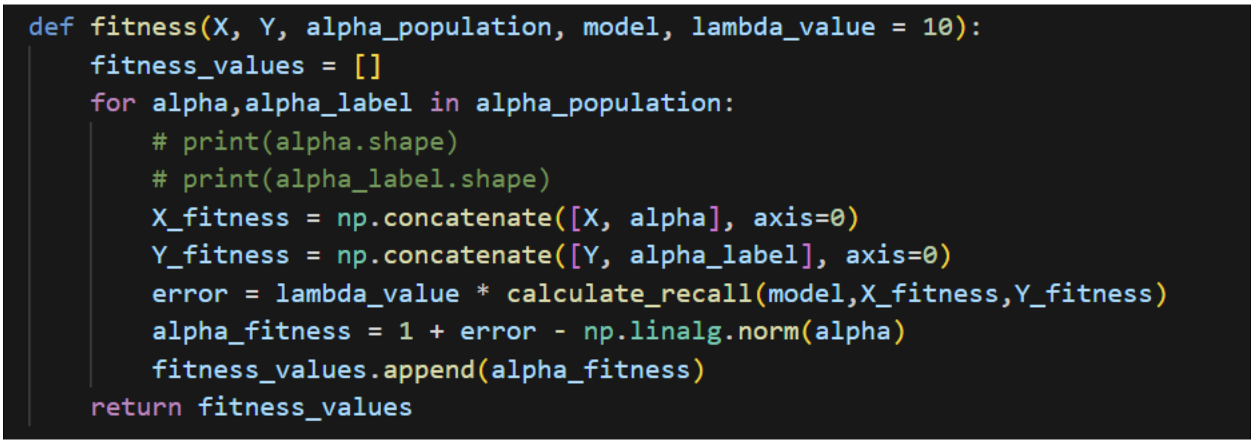


```
def anneal(alpha, mask_a, d=20, lower_bound=2, upper_bound=10):
    alpha = alpha.copy()
    mask_b = np.random.choice([True, False], size=alpha.shape)
    mask = mask_a ^ mask_b
    step = np.random.randint(-d, d+1, size=alpha.shape)/225.
    start_h = np.random.randint(0, lower_bound)
    end_h = np.random.randint(alpha.shape[0] - upper_bound, alpha.shape[0])
    start_w = np.random.randint(0, lower_bound)
    end_w = np.random.randint(alpha.shape[1] - upper_bound, alpha.shape[1])
    masksliced = np.zeros(alpha.shape, dtype=bool)
    masksliced[start_h:end_h, start_w:end_w] = mask[start_h:end_h, start_w:end_w]
    # print(masksliced.shape)
    # print(alpha.shape)
    alpha[masksliced] += step[masksliced]
    return alpha
```



Fitness Function-

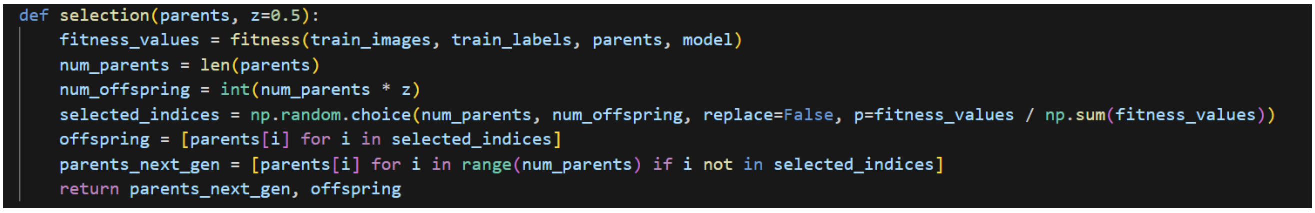
The fitness function calculates fitness values for a population of candidate solutions (alpha_population) in a genetic algorithm context. It assesses each candidate solution's fitness based on its ability to improve the performance of a given machine learning model (model) when combined with the original dataset (X and Y). The function iterates through the population, concatenates each candidate solution with the original dataset, and evaluates the resulting dataset's performance using a specified evaluation metric (here, calculate_recall). The fitness value for each candidate solution is computed as a combination of the model's performance improvement and the norm of the candidate solution, adjusted by a regularization parameter (lambda_value). This process guides the genetic algorithm towards selecting candidate solutions that enhance the model's performance while controlling the complexity of the solutions.



```
def fitness(X, Y, alpha_population, model, lambda_value = 10):
    fitness_values = []
    for alpha,alpha_label in alpha_population:
        # print(alpha.shape)
        # print(alpha_label.shape)
        X_fitness = np.concatenate([X, alpha], axis=0)
        Y_fitness = np.concatenate([Y, alpha_label], axis=0)
        error = lambda_value * calculate_recall(model,X_fitness,Y_fitness)
        alpha_fitness = 1 + error - np.linalg.norm(alpha)
        fitness_values.append(alpha_fitness)
    return fitness_values
```

Selection Function-

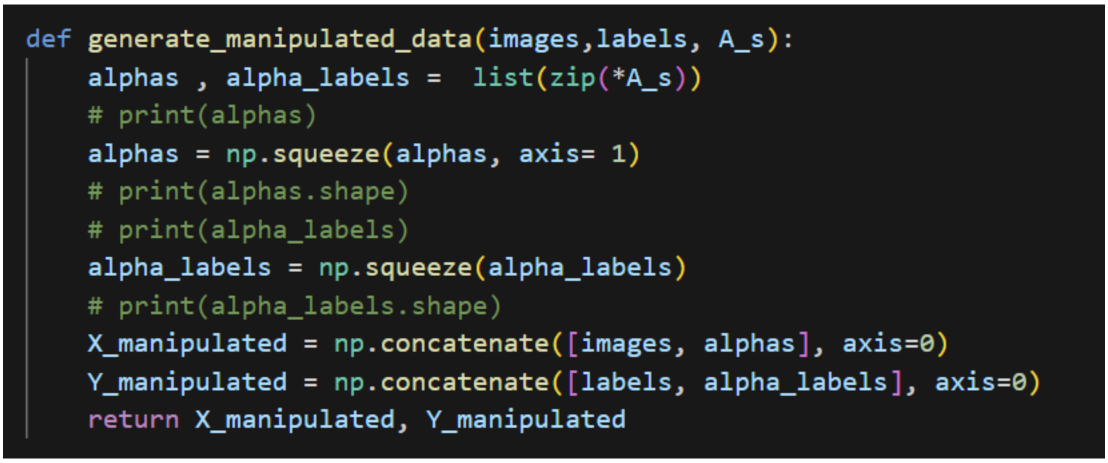
The selection function performs selection in a genetic algorithm to determine which candidate solutions from the parent population are chosen to produce offspring for the next generation. It calculates the fitness values for each candidate solution in the parent population based on their performance when combined with the training dataset and a given model. Then, it selects a proportion (z) of the parent population to produce offspring. The selection process is probabilistic, with each candidate solution's probability of being selected proportional to its fitness value relative to the sum of fitness values across all candidates. After selecting the offspring, the function returns the remaining parent candidates for the next generation along with the selected offspring. This process ensures that candidate solutions with higher fitness values, indicating better performance, have a higher chance of being chosen for the next generation, thus guiding the evolutionary process towards potentially more promising solutions.



```
def selection(parents, z=0.5):
    fitness_values = fitness(train_images, train_labels, parents, model)
    num_parents = len(parents)
    num_offspring = int(num_parents * z)
    selected_indices = np.random.choice(num_parents, num_offspring, replace=False, p=fitness_values / np.sum(fitness_values))
    offspring = [parents[i] for i in selected_indices]
    parents_next_gen = [parents[i] for i in range(num_parents) if i not in selected_indices]
    return parents_next_gen, offspring
```

Generate Data Function-

The generate_manipulated_data function generates manipulated data by combining original images (images) and their corresponding labels (labels) with a set of additional data points (A_s). These additional data points are represented by alpha arrays (alphas) and their corresponding labels (alpha_labels). The function first extracts the alpha arrays and their labels from A_s, then concatenates them with the original images and labels along the data axis. The resulting manipulated dataset, consisting of both original and additional data points, is returned as X_manipulated and Y_manipulated, respectively. This function facilitates the augmentation of the original dataset with additional data points, which can be useful for tasks such as data augmentation in machine learning.



```
def generate_manipulated_data(images,labels, A_s):
    alphas , alpha_labels =  list(zip(*A_s))
    # print(alphas)
    alphas = np.squeeze(alphas, axis= 1)
    # print(alphas.shape)
    # print(alpha_labels)
    alpha_labels = np.squeeze(alpha_labels)
    # print(alpha_labels.shape)
    X_manipulated = np.concatenate([images, alphas], axis=0)
    Y_manipulated = np.concatenate([labels, alpha_labels], axis=0)
    return X_manipulated, Y_manipulated
```

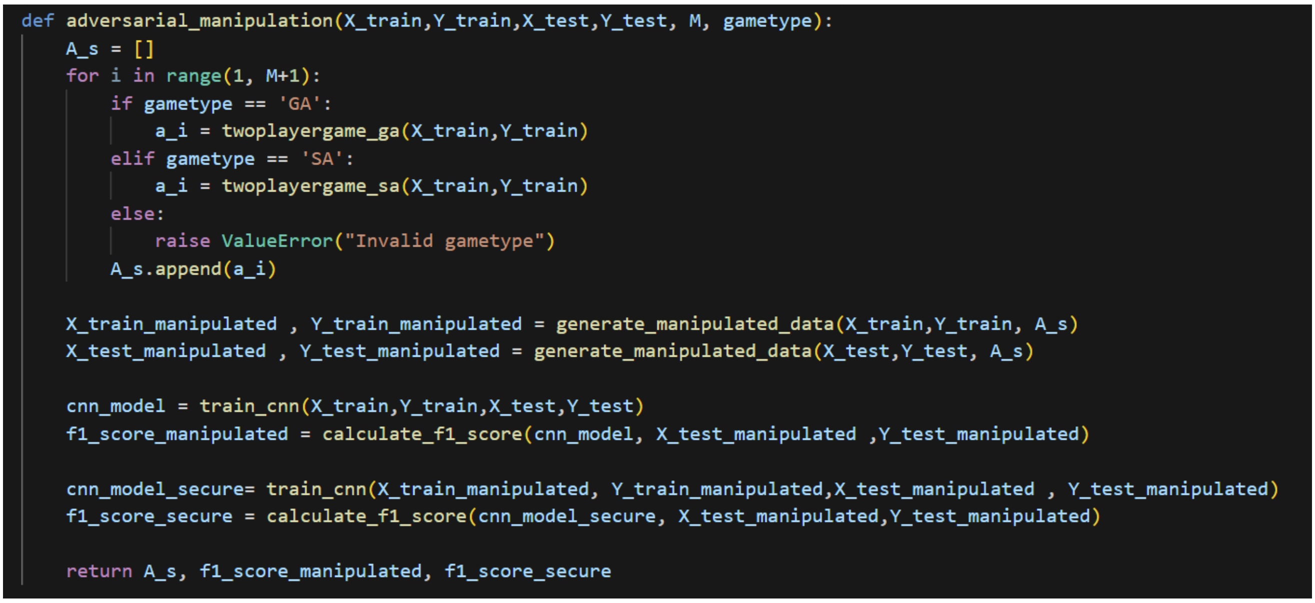
Implementation of Main Algorithms

Main Function-

The adversarial_manipulation function performs adversarial manipulation on the training and testing datasets (X_train, Y_train, X_test, Y_test) using a specified game type (gametype) for a given number of iterations (M). It iteratively generates adversarial data points (A_s) by playing a two-player game based on the chosen game type (GA for Genetic Algorithm or SA for Simulated Annealing). These adversarial data points are then used to generate manipulated versions of the training and testing datasets using the generate_manipulated_data function.

After generating the manipulated datasets, the function trains a convolutional neural network (CNN) model (cnn_model) on the original training data and evaluates its performance on the manipulated testing data to calculate the F1 score (f1_score_manipulated). Subsequently, it trains another CNN model (cnn_model_secure) on the manipulated training data and evaluates its performance on the same manipulated testing data to calculate the F1 score (f1_score_secure).

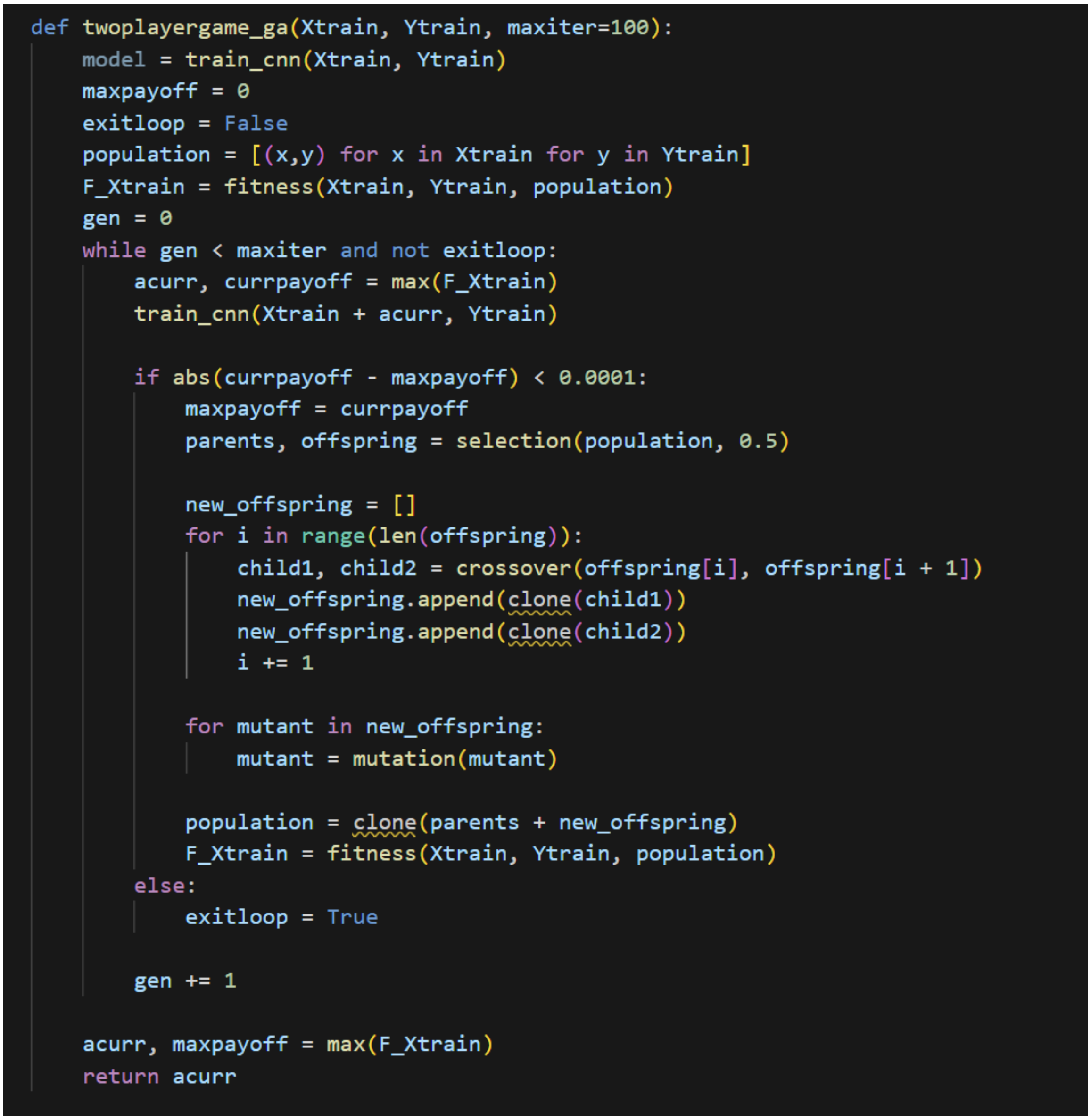
Finally, the function returns the adversarial data points (A_s), the F1 score of the CNN model trained on the manipulated testing data (f1_score_manipulated), and the F1 score of the secure CNN model trained on the manipulated training and testing data (f1_score_secure). This function serves to assess the impact of adversarial manipulation on the CNN model's performance and evaluate the effectiveness of securing the model against such manipulations.



```
def adversarial_manipulation(X_train,Y_train,X_test,Y_test, M, gametype):
    A_s = []
    for i in range(1, M+1):
        if gametype == 'GA':
            a_i = twoplayergame_ga(X_train,Y_train)
        elif gametype == 'SA':
            a_i = twoplayergame_sa(X_train,Y_train)
        else:
            raise ValueError("Invalid gametype")
        A_s.append(a_i)

    X_train_manipulated , Y_train_manipulated = generate_manipulated_data(X_train,Y_train, A_s)
    X_test_manipulated , Y_test_manipulated = generate_manipulated_data(X_test,Y_test, A_s)

    cnn_model = train_cnn(X_train,Y_train,X_test,Y_test)
    f1_score_manipulated = calculate_f1_score(cnn_model, X_test_manipulated ,Y_test_manipulated)

    cnn_model_secure= train_cnn(X_train_manipulated, Y_train_manipulated,X_test_manipulated , Y_test_manipulated)
    f1_score_secure = calculate_f1_score(cnn_model_secure, X_test_manipulated,Y_test_manipulated)

    return A_s, f1_score_manipulated, f1_score_secure
```

First Algorithm Rough Code-

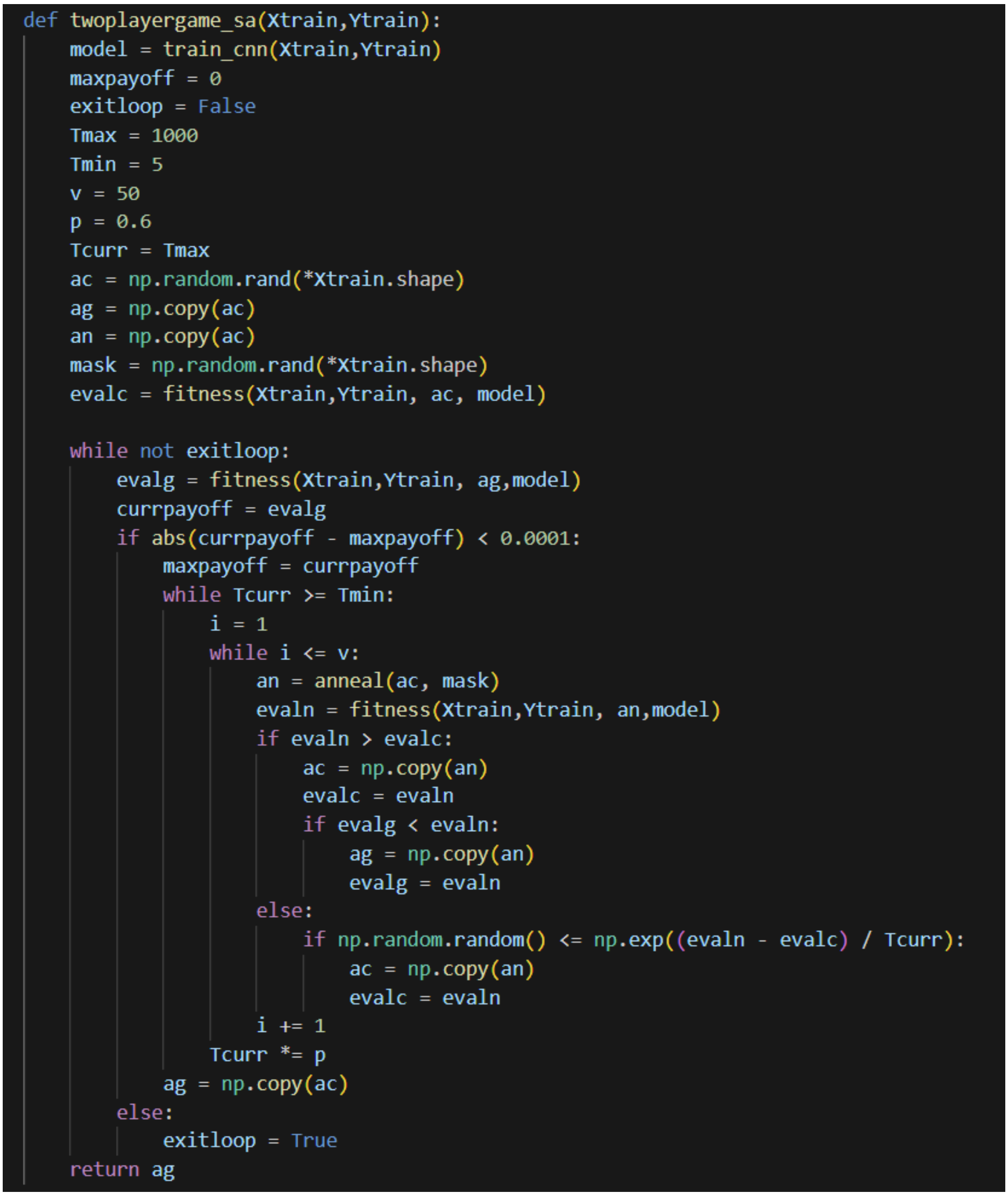
The twoplayergame_ga function trains a Convolutional Neural Network (CNN) model on a given training dataset (Xtrain, Ytrain) using a genetic algorithm (GA) based two-player game. It initializes a population of candidate solutions, evaluates their fitness, and iteratively evolves them to improve the CNN's performance. Each iteration involves selecting parents based on fitness, performing crossover and mutation operations to produce offspring, and updating the population. The process continues until a maximum number of iterations (maxiter) is reached or the performance improvement becomes negligible. Finally, the function returns the most effective adversarial data point (acurr) discovered during the evolution process.



```python
def twoplayergame_ga(Xtrain, Ytrain, maxiter=100):
    model = train_cnn(Xtrain, Ytrain)
    maxpayoff = 0
    exitloop = False
    population = [(x,y) for x in Xtrain for y in Ytrain]
    F_Xtrain = fitness(Xtrain, Ytrain, population)
    gen = 0
    while gen < maxiter and not exitloop:
        acurr, currpayoff = max(F_Xtrain)
        train_cnn(Xtrain + acurr, Ytrain)

        if abs(currpayoff - maxpayoff) < 0.0001:
            maxpayoff = currpayoff
            parents, offspring = selection(population, 0.5)

            new_offspring = []
            for i in range(len(offspring)):
                child1, child2 = crossover(offspring[i], offspring[i + 1])
                new_offspring.append(clone(child1))
                new_offspring.append(clone(child2))
                i += 1

            for mutant in new_offspring:
                mutant = mutation(mutant)

            population = clone(parents + new_offspring)
            F_Xtrain = fitness(Xtrain, Ytrain, population)
        else:
            exitloop = True

        gen += 1

    acurr, maxpayoff = max(F_Xtrain)
    return acurr
```

Second Algorithm Rough Code-

The twoplayergame_sa function implements a two-player game using simulated annealing (SA) to train a Convolutional Neural Network (CNN) model on a given training dataset (Xtrain, Ytrain). It initializes annealing parameters such as temperature (Tmax, Tmin), cooling rate (p), and the number of iterations (v). The function starts with a randomly initialized adversarial data point (ac) and iteratively improves it by annealing, evaluating fitness, and updating the solution based on the Metropolis criterion. The process continues until the performance improvement becomes negligible or a stopping criterion is met. Finally, the function returns the adversarial data point (ag) with the highest fitness, representing an effective manipulation for the CNN model.



```python
def twoplayergame_sa(Xtrain,Ytrain):
    model = train_cnn(Xtrain,Ytrain)
    maxpayoff = 0
    exitloop = False
    Tmax = 1000
    Tmin = 5
    v = 50
    p = 0.6
    Tcurr = Tmax
    ac = np.random.rand(*Xtrain.shape)
    ag = np.copy(ac)
    an = np.copy(ac)
    mask = np.random.rand(*Xtrain.shape)
    evalc = fitness(Xtrain,Ytrain, ac, model)

    while not exitloop:
        evalg = fitness(Xtrain,Ytrain, ag,model)
        currpayoff = evalg
        if abs(currpayoff - maxpayoff) < 0.0001:
            maxpayoff = currpayoff
            while Tcurr >= Tmin:
                i = 1
                while i <= v:
                    an = anneal(ac, mask)
                    evaln = fitness(Xtrain,Ytrain, an,model)
                    if evaln > evalc:
                        ac = np.copy(an)
                        evalc = evaln
                        if evalg < evaln:
                            ag = np.copy(an)
                            evalg = evaln
                    else:
                        if np.random.random() <= np.exp((evaln - evalc) / Tcurr):
                            ac = np.copy(an)
                            evalc = evaln
                    i += 1
                Tcurr *= p
            ag = np.copy(ac)
        else:
            exitloop = True
    return ag
```

Conclusion

We have formulated a maxmin problem for adversarial learning with both two-player sequential games and multiplayer stochastic games over deep learning networks. The experiments demonstrate the correctness and performance of proposed adversarial algorithm. The algorithm converges onto adversarial manipulations affecting testing performance in deep learning networks. This allows us to propose a secure learner that is immune to the adversarial attacks on deep learning. We have shown that our model is significantly more robust than traditional CNN and GAN under adversarial attacks. By changing the game formulation, we can experiment with adversarial payoff functions over randomized strategy spaces. The attack scenarios over such strategy spaces would determine multiplayer games over mixed strategies. In the future, we plan to investigate more challenging scenarios where adversaries attack multiple labels simultaneously.

References

- A. S. Chivukula and W. Liu, "Adversarial Deep Learning Models with Multiple Adversaries," in IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 6, pp. 1066-1079, 1 June 2019, doi: 10.1109/TKDE.2018.2851247.