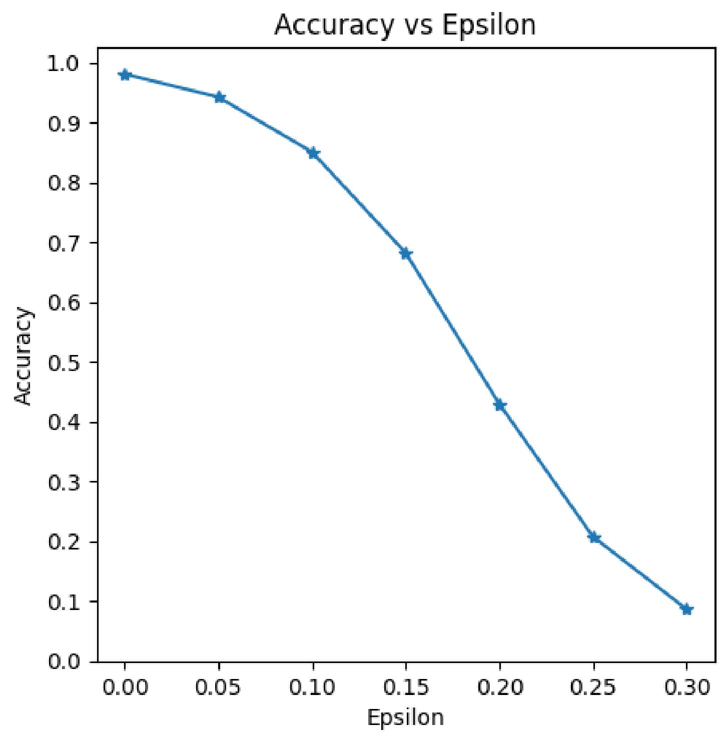
Moellering_Matthew_Project3

1.1

Graph



Image

1.2

Yes the model can not confirm at least epsilon for each pixel based off the following code:

```
def fgsm_attack(image, epsilon, data_grad):
    # !! Put your code below
    # Collect the element-wise sign of the data gradient, you can use data_grad.sign()
    attack = image +epsilon*data_grad.sign()
    # Create the perturbed image by adjusting each pixel of the input image
    pert_image = torch.clamp(attack, 0, 1)
```

```
    # Adding clipping to maintain [0,1] range, you can use function torch.clamp

    # Return the perturbed image
    return pert_image
    # !! Put your code above
```

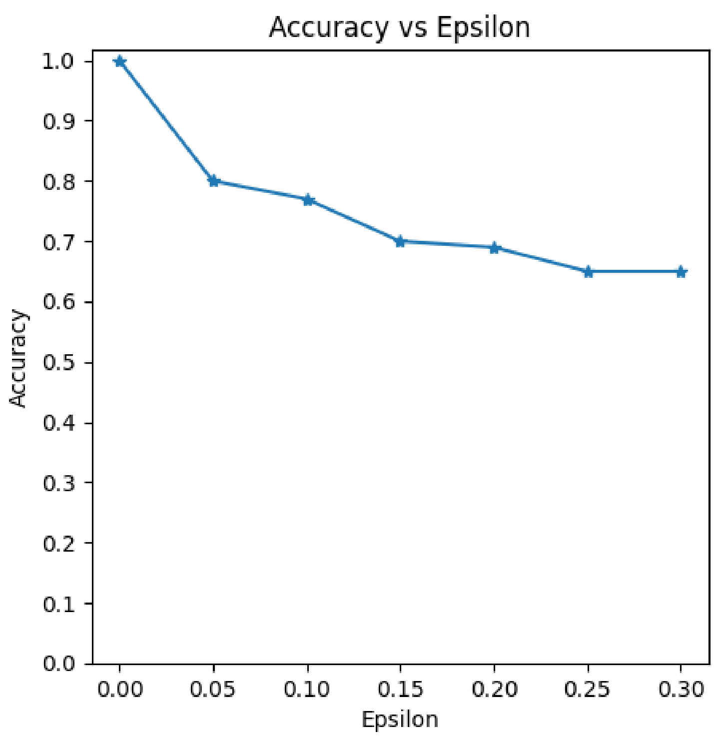
1.3

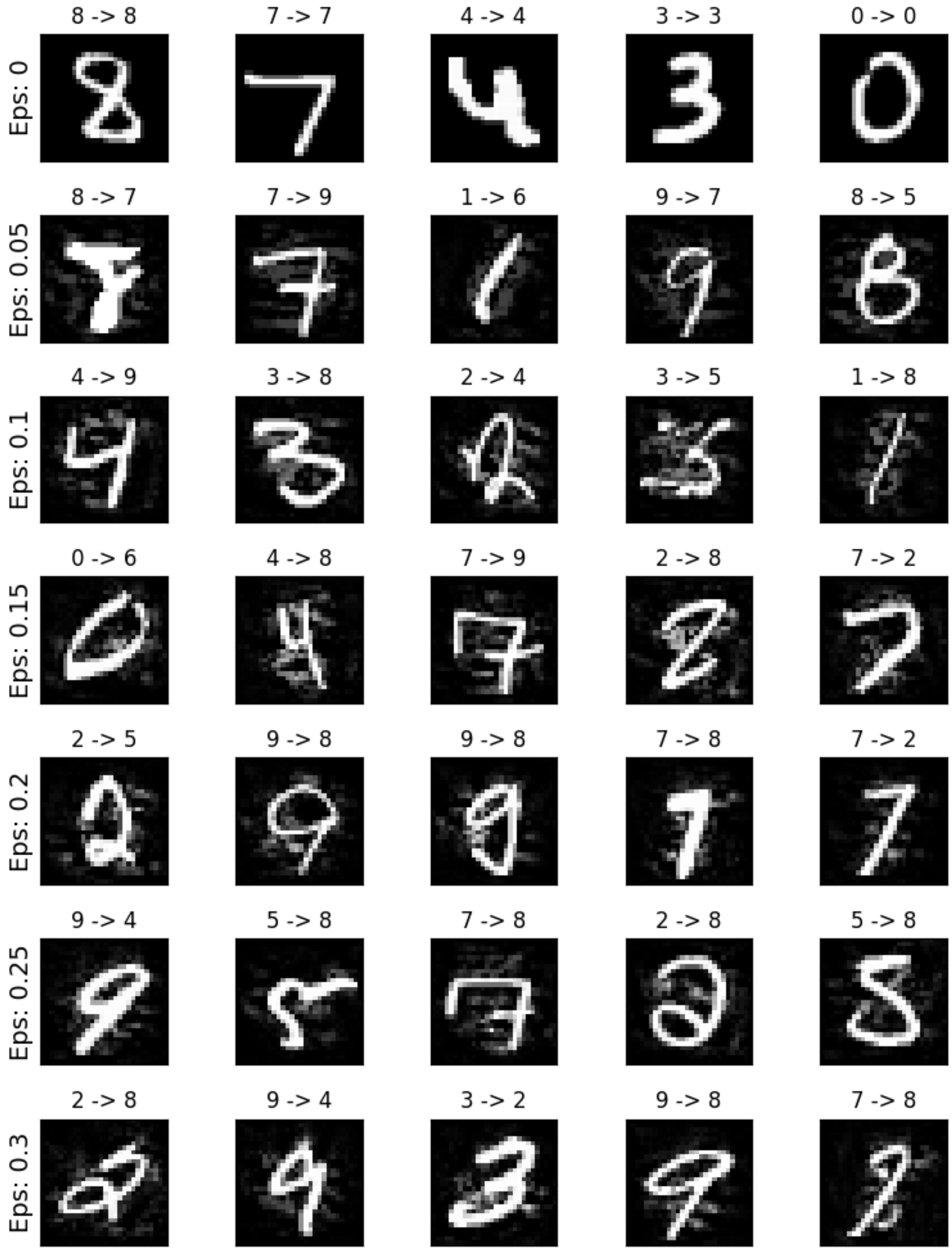
Strictly lesser, the advanced function will always be greater. The epsilon value will continue to increase and decrease loss therefore decreasing performance as the model continues to train.

1.4

The relationship will be weakly greater. The function is still increasing as it adds the gradient but it will be at a much slower rate than that of the previous value.

2.1



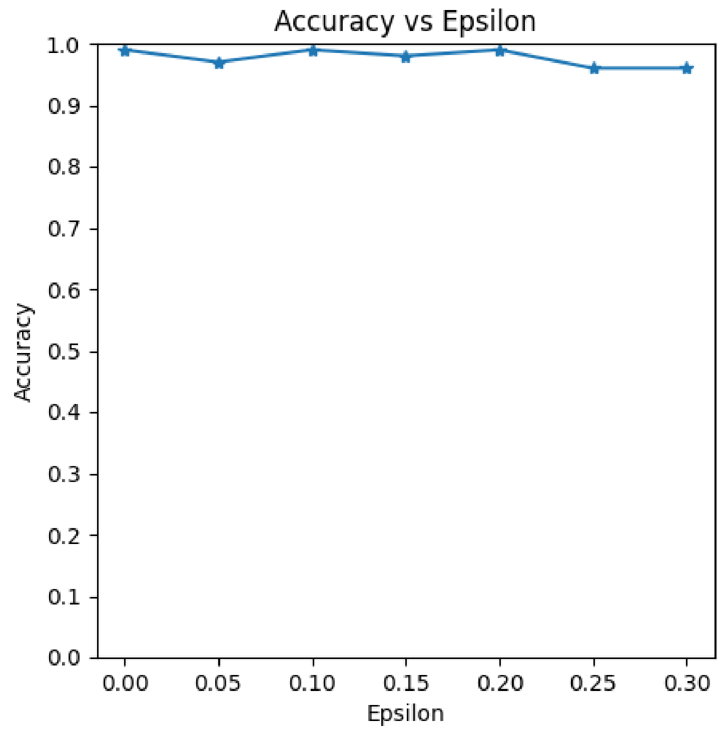


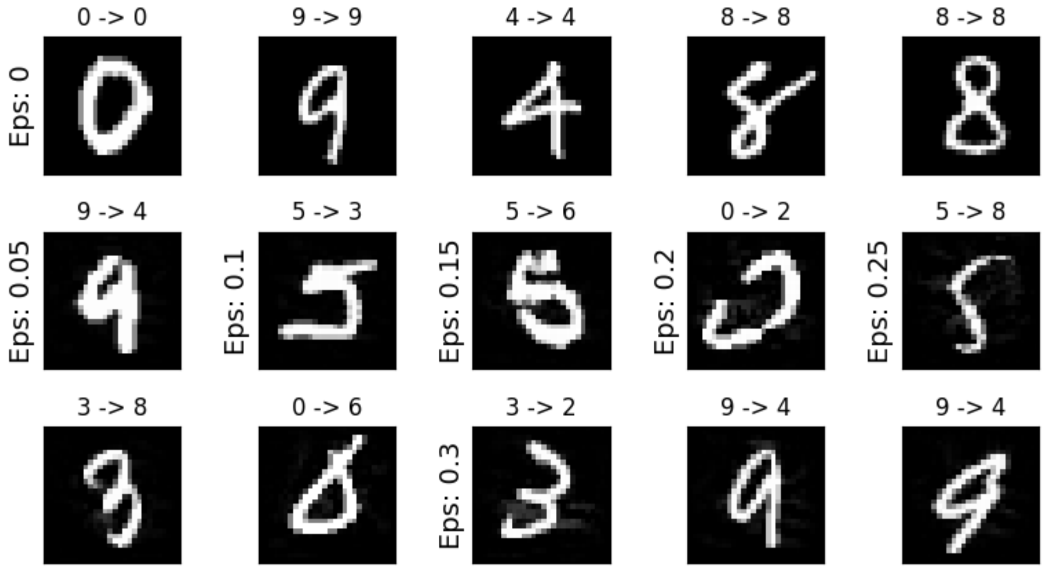
2.2

FGSM is faster but PGD is less obvious and therefore more useful, and can degrade a model for a longer time without being picked up by other algorithms. PGD's results look more natural while FGSM has clearly been tampered with.

2.3

Add the image back into the delta to bring it closer to the loss of the image.


Accuracy vs Epsilon
Accuracy
Epsilon
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.00
0.05
0.10
0.15
0.20
0.25
0.30



IT did not work.

3.1

My code had errors and I could not get it to work below is the logic I was trying to code

```
population = torch.empty([N] + dims, device=device).uniform_(-epsilon, epsilon)
    population = torch.clamp(population + image, 0, 1) - image
    # if you prefer to use a list, you may consider the following
    #population = []
    #for n in range(N):
    #    rand_image = torch.empty(dims, device=device).uniform_(-epsilon, epsilon)
    #    rand_image = torch.clamp(rand_image + image, 0, 1) - image
    #    population.append(rand_image)

    # initialize two parameters rho=0.5 and beta=0.4
    rho = 0.5
    beta = 0.4
    # initialize num_plateaus to be 0
    num_plateaus = 0
    num_iter =1
    previous_elite_score = -1000
    for i in range(num_iter):
        # For each member in the current population, compute the fitness score. Note that you will need to clamp the
        # value to a large range, e.g., [-1000,1000] to avoid getting "inf"

        fitness_score = torch.clamp(torch.log(population)-torch.log(sum(population)), -1000, 1000)
        # Find the elite member, which is the one with the highest fitness score
        elite_member = torch.max(fitness_score)
        # Add the elite member to the new population
        population = population + elite_member
        # If the elite member can succeed in attack, terminate and return the elite member
        if torch.argmax(elite_member) != target_class:
            return elite_member
        # If the elite member’s fitness score is no better than the last population’s elite member’s fitness score,
        # increment num_plateaus. It is recommended to use a threshold of 1e-5 to avoid numerical instability
        if elite_member - previous_elite_score <= 1e-5:
            num_plateaus += 1
        previous_elite_score = elite_member
        # Compute the probability each member in the population should be chosen by applying softmax to the fitness
        # scores
        probs = torch.softmax(fitness_score, dim = 0)
        # Choose a member in the current population according to the probability, name it parent_1
        parent_1 = torch.sample(population)
        # Choose a member in the current population according to the probability, name it parent_2
        parent_2 = torch.sample(population)
        # Generate a “child” image from parent1 and parent2: For each pixel, take parent1’s corresponding pixel
        # value with probability p=fitness(parent1)/(fitness(parent1)+fitness(parent2))
        # and take parent2’s corresponding pixel value with probability 1-p

        child = torch.gather(torch.stack((parent_1, parent_2)), 0, probs)
        # With probability q, add a random noise to the children image with pixel-wise value uniformly sampled from
        # [-beta*epsilon,beta*epsilon]
        noise = torch.empty(1, parent_1.shape[0]).uniform_(-beta*epsilon, beta*epsilon).to(device)
        # Apply clipping on the child image to make sure it is in the feasible region F
        child = torch.clamp(noise+image, 0, 1)
  # Add this child to the population, repeat generating children in this way until the population has N members
        nextPopulation = torch.empty((population.shape[0], population.shape[1]))

        for children in range(N):
            nextPopulation[children] = child
        # Update the value of rho as max(rho_min,0.5*0.9^num_plateaus)
        rho = max(rho_min, 0.5*0.9**num_plateaus)
        # Update the value of beta as max(beta_min,0.4*0.9^num_plateaus)
        beta = max(beta_min,0.4*0.9^num_plateaus)
        # !! Put your code above

    print(perturbed_image.shape)
    perturbed_image = nextPopulation
    # Return the perturbed image
    return perturbed_image
```

3.2

Include the amount of children in the population. That way it increases the amount of errors.

3.3