Adversarial Attacks on Neural Networks

By Christo-Odysseus Keramitzis

1. Introduction

This report will explore the effect of Adversarial attacks and their implications on generating adversarial objects that reduce accuracy in neural networks, specifically convolutional neural networks(CNN). This will be shown through the analysis of targeted Fast gradient sign method (T-FGSM) attacks and the corresponding defence of adversarial training to detect the perturbed data. Adversarial attacks aim at generating false input perceived as normal. The purpose is to create noise that will over time degrade the accuracy of the algorithm. This is done through perturbations of input gradients that slowly corrupt the algorithm's decisions. The influence gradients of data has upon standard accuracy will be a primary focus including how small gradient and parameter changes gradually alter the functioning of the CNN model. As this subject is becoming increasingly prevalent in society as ad-hoc and mass adoption of artificial intelligence throughout all aspects of life is becoming increasingly common; in addition to its rapid development, the game of tug between bad actors against artificial intelligence and security specialist is becoming more uncertain as new methods of attack and mitigations are continuously being developed. The aim is to achieve a holistic understanding of the fragility of neural networks and the necessity to harden their security against adversarial attacks.

2. Reasonable solution on the project

The CNN will use a preconfigured data set that consists of classes ‘0’ to ‘9’. This dataset consists of 6000 training and 1000 test examples that the model can implement and iterate through to detect numbers. Each dataset used is loaded into batches of 128 samples in size for better usability and speed when iterating.

```
train_loader = torch.utils.data.DataLoader(
    train_data, batch_size=128, shuffle=True, num_workers=2)

test_loader = torch.utils.data.DataLoader(
    test_data, batch_size=128, shuffle=True, num_workers=2)

classes = ('zero', 'one', 'two', 'three', 'four',
           'five', 'six', 'seven', 'eight', 'nine')
```

```
==> Loading data..
6000
1000
```

The model

A brief description on how the targeted model functions will be given for context and understanding of how fgsm targets the classes used to detect numbers. Multilayer perceptron classifier that is used follows a specific structure that the model follows.MLP consists of an input layer that consists of nodes representing extracted features, 4 hidden layers where the calculations are performed and epochs generated to produce an output for the output layer. The output layer generates the final predictions using the processed parameters from the hidden layer. As the model follows and uses Multilayer perceptron architecture due to the classifier used, the structure of the Model is as follows:

1. Initialization

The model is initialised with weight nodes and bias with a Multilayer perceptron architecture in mind. During the initialization phase, the three features are initialised by allocating an input and output sample. It applies a linear transformation through this function (figure 1):

$$: y = xA^T + b.$$ (Linear — PyTorch 1.8.0 documentation. (n.d.))

This is initialised within a classification class as depicted in figure 2 that uses MLP classifier and Relu as the activation function. Rectified linear unit (ReLu) is the most widely used activation function for CNN models due to its speed and adaptiveness to work with values greater and lower than 0. Furthermore, in order to properly use the stochastic gradient descent (SGD) as an optimiser during backpropagation; an activation function that acts in a linear fashion is required.

```
[ ]  class Classifier(torch.nn.Module):
         def __init__(self):
             super(Classifier,self).__init__()
             self.fc1 = torch.nn.Linear(28*28,512)
             self.fc2 = torch.nn.Linear(512,128)
             self.fc3 = torch.nn.Linear(128,10)
```

Figure 2

2. Forward propagation

ReLus ability to act in a linear fashion while being non-linear is its greatest advantage as it allows for better optimisation functions such as SGD rather than tahn or sigmoid. During forward propagation, the inputs are passed through the network that applies the activation function to the weights of the features stated. Relu works as a simple ‘on or off’ switch; all values greater than 0 are returned normally but those less than are defaulted to zero. This allows the use of gradient based methods while also preserving the effectiveness of linear models. This is directly implemented within the classifier class as depicted in figure 3; the fc parameters are each called and activated thus creating a level of nonlinearity.

```
class Classifier(torch.nn.Module):
    def __init__(self):
        super(Classifier,self).__init__()
        self.fc1 = torch.nn.Linear(28*28,512)
        self.fc2 = torch.nn.Linear(512,128)
        self.fc3 = torch.nn.Linear(128,10)

    def forward(self,x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))

        return x
```

Figure 3

3. Loss function

Loss functions are an important element in increasing the performance and effectiveness of Neural networks. Their purpose is to ensure the gradients/predictions do not deviate too far from the initial base or goal. Cross Entropy allows the difference between these two values to be measured through the following equation:

$$-\sum_{c=1}^{N} y_c log(p_c)$$

Categorical cross-entropy for a single instance ((2024, January). Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy [Review of Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy]. Datacamp.com; Kurtis pykes.)

Due to the multiple classes, a categorical cross entropy function is used where each class's entropy is calculated and summed. As shown in figure 4 variable criterion is declared using cross entropy which will later become a used value by FGSM to perturbate the gradients; emphasising the importance of functions that impact the gradient.

```
# Model
print('==> Building model..')
# Initialize the classifier
net = Classifier()
# Define the learning rate
lr = 0.001
# Define a loss function
criterion = nn.CrossEntropyLoss()
```

Figure 4

4. Backpropagation

Backpropagation is where the weights and biases are perfected and fine tuned to be as accurate as possible. This is achieved through gradient lost functions, learning rates, loss functions and other gradient tuning functions. This model utilises a learning rate of 0.001 and the SGD optimiser with momentum of 0.9 and decay of 5e-4 depicted in figure 6. The learning rate as seen in figure 5 is used within the SGD optimisation algorithm that impacts the ‘step size’ or the value each gradient is updated with. A balancing act has to be made when choosing the learning rate value. The recommended value is between 0 and 1, a large Lr can cause the gradient to develop too quickly making it ‘overshoot’ with too many epochs; while a smaller one can develop too little.

```
# Define the learning rate
lr = 0.001
```

Figure 5

The reason SGD is used rather than traditional gradient descent is due to the large dataset. SGD works by randomly selecting any data points 1 through n and calculating loss individually rather than iterating through every data point and calculating the sum of them all. Momentum is an optimisation parameter that takes into account previous changes to optimisation that aims to bring it closer to the desired goal. This aids in improving the speed

of calculation by adding the momentum to gradient changes. The purpose if weight decay is the opposite to momentum's, that is, it reduces stray gradients in comparison to the general sum gradient to aid in feature sharing, prevent overfitting, ensure smaller gradients are being trained and improves generalisation of the optimiser. This also prevents the model from continuously defaulting the parameters to 0 from a large loss. Due to the small learning rate, a low weight decay is selected to prevent underfitting.

```
# Select a optimization method
optimizer = optim.SGD(net.parameters(), lr=lr,
                      momentum=0.9, weight_decay=5e-4)
```

Figure 6

5. Training

The optimisation methods, forward and backward propagation and initialisation parameters are iterated through a test and train function n(10) amount of times to reach final output nodes; which represents the decision the model makes and its subsequent standard accuracy. The train class depicted in figure 7 shows how each parameter will be iterated through and what variables are initialised to generate a gradient. The optimiser first zeros all parameter gradients to ensure generalisation and equality between all data points that aids in reducing under and over fitting.

```
# Training
def train(epoch, train_loader):
    print('\nEpoch: %d' % epoch)
    net.train()
    train_loss = 0

    for batch_idx, (inputs, targets) in enumerate(train_loader):
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

        # print statistics
        if batch_idx % 30 == 0:
            print('Train Epoch: {} [{}/{}]\tLoss: {:.6f}'.format(epoch, batch_idx * len(inputs),
                                                                 len(train_loader.dataset), loss.item()))
    return train_loss
```

Figure 7

The following figure shows the test dataset which will be used to evaluate the training data. The primary difference between the datasets is that the CNN will not learn from the test data as it is there for validation purposes. the two primary methods of testing are either after every epoch is completed or after the final epoch. As this is a smaller dataset the test is done after every epoch.

```python
# Test
def test(epoch, test_loader):
    net.eval()
    test_loss = 0
    correct = 0 # number of correct predictions
    total = 0 # number of all test examples

    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(test_loader):
            # calculate outputs by running images through the network
            outputs = net(inputs)
            loss = criterion(outputs, targets)
            test_loss += loss.item()
            # the class with the highest probability is what we choose as prediction
            _, predicted = outputs.max(1)

            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
    acc = 100 * correct / total
    # print statistics
    print('Train Epoch: {}, Accuracy: {:.6f}%'.format(epoch, acc))
    return test_loss, acc
```

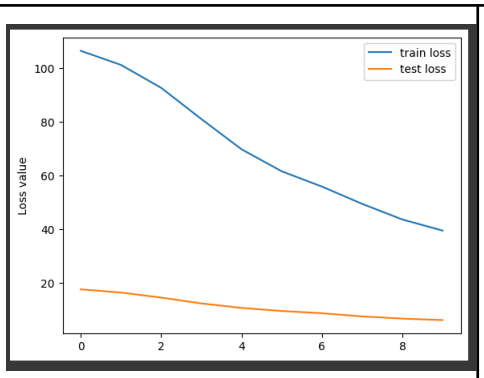
Figure 8

The Datasets are looped over 10 times and the results are appended and depicted for better reading and analysis in figure 9.

```python
[ ]  train_losses = []
     test_losses = []

     # # loop over the dataset multiple times
     for epoch in range(0,10):
         trl = train(epoch, train_loader)
         train_losses.append(trl)
         tstl, acc = test(epoch, test_loader)
         test_losses.append(tstl)

     print('Accuracy of the network on the test images: %d %%' % (acc))
```

Figure 9

Final results:




```
Epoch: 0
Train Epoch: 0 [0/6000] Loss: 2.302234
Train Epoch: 0 [3840/6000]      Loss: 2.262031
Train Epoch: 0, Accuracy: 45.200000%

Epoch: 1
Train Epoch: 1 [0/6000] Loss: 2.222993
Train Epoch: 1 [3840/6000]      Loss: 2.147308
Train Epoch: 1, Accuracy: 50.600000%

Epoch: 2
Train Epoch: 2 [0/6000] Loss: 2.019761
Train Epoch: 2 [3840/6000]      Loss: 1.943141
Train Epoch: 2, Accuracy: 56.500000%

Epoch: 3
Train Epoch: 3 [0/6000] Loss: 1.912028
Train Epoch: 3 [3840/6000]      Loss: 1.771532
Train Epoch: 3, Accuracy: 63.000000%

Epoch: 4
Train Epoch: 4 [0/6000] Loss: 1.614347
Train Epoch: 4 [3840/6000]      Loss: 1.485608
Train Epoch: 4, Accuracy: 68.500000%

Epoch: 5
Train Epoch: 5 [0/6000] Loss: 1.539598
Train Epoch: 5 [3840/6000]      Loss: 1.258721
Train Epoch: 5, Accuracy: 70.000000%

Epoch: 6
Train Epoch: 6 [0/6000] Loss: 1.129210
Train Epoch: 6 [3840/6000]      Loss: 1.161818
Train Epoch: 6, Accuracy: 72.100000%

Epoch: 7
Train Epoch: 7 [0/6000] Loss: 1.249073
Train Epoch: 7 [3840/6000]      Loss: 0.886604
Train Epoch: 7, Accuracy: 76.200000%

Epoch: 8
Train Epoch: 8 [0/6000] Loss: 0.966252
Train Epoch: 8 [3840/6000]      Loss: 0.992912
Train Epoch: 8, Accuracy: 78.300000%

Epoch: 9
Train Epoch: 9 [0/6000] Loss: 0.840200
Train Epoch: 9 [3840/6000]      Loss: 0.935559
Train Epoch: 9, Accuracy: 79.300000%
Accuracy of the network on the test images: 79 %
```

6. Prediction

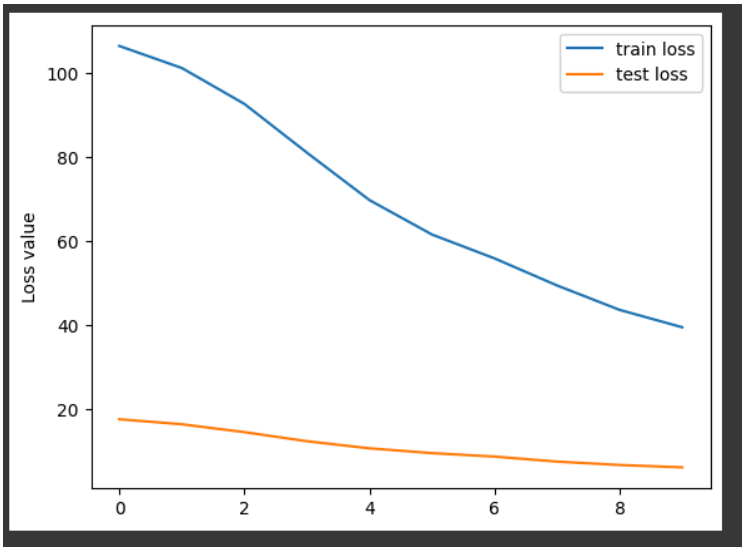
The algorithm, having iterated through the datasets and parameters. converges on a final prediction through many output nodes. This output layer typically can have 1 node, for binary decisions or many that represent each class. Each node present within the output layer represents class 0-9 that are paired with a distinct probability obtained through the input and hidden layer. to gain a better understanding of the influence individual parameters can have on the accuracy and efficiency of neural networks; adjustable parameters will be altered and depicted in the following tables.

Effect of metric changes

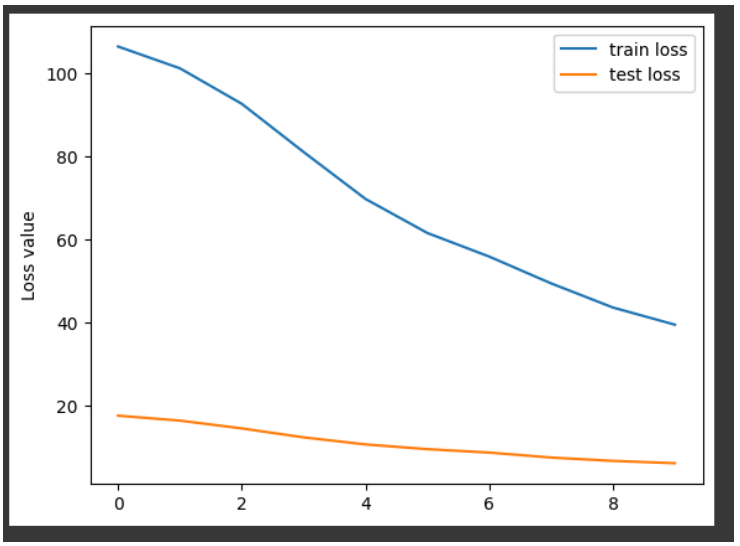
Weight decay = 5e-4	Weight decay = 1e-4

```
print("Predicted labels:",samples_pred)
```

```
Original labels: tensor([0, 2, 3, 4, 5, 6, 7, 8])
Predicted labels: tensor([0, 1, 3, 4, 5, 6, 7, 8])
```



```
Original labels: tensor([0, 2, 3, 4, 5, 6, 7, 8])
Predicted labels: tensor([0, 1, 5, 4, 5, 6, 7, 5])
```



```
Epoch: 0
Train Epoch: 0 [0/6000] Loss: 2.302234
Train Epoch: 0 [3840/6000]      Loss: 2.262031
Train Epoch: 0, Accuracy: 45.200000%

Epoch: 1
Train Epoch: 1 [0/6000] Loss: 2.222993
Train Epoch: 1 [3840/6000]      Loss: 2.147308
Train Epoch: 1, Accuracy: 50.600000%

Epoch: 2
Train Epoch: 2 [0/6000] Loss: 2.019761
Train Epoch: 2 [3840/6000]      Loss: 1.943141
Train Epoch: 2, Accuracy: 56.500000%

Epoch: 3
Train Epoch: 3 [0/6000] Loss: 1.912028
Train Epoch: 3 [3840/6000]      Loss: 1.771532
Train Epoch: 3, Accuracy: 63.000000%

Epoch: 4
Train Epoch: 4 [0/6000] Loss: 1.614347
Train Epoch: 4 [3840/6000]      Loss: 1.485608
Train Epoch: 4, Accuracy: 68.500000%

Epoch: 5
Train Epoch: 5 [0/6000] Loss: 1.539598
Train Epoch: 5 [3840/6000]      Loss: 1.258721
Train Epoch: 5, Accuracy: 70.000000%

Epoch: 6
Train Epoch: 6 [0/6000] Loss: 1.129210
Train Epoch: 6 [3840/6000]      Loss: 1.161818
Train Epoch: 6, Accuracy: 72.100000%

Epoch: 7
Train Epoch: 7 [0/6000] Loss: 1.249073
Train Epoch: 7 [3840/6000]      Loss: 0.886604
Train Epoch: 7, Accuracy: 76.200000%

Epoch: 8
Train Epoch: 8 [0/6000] Loss: 0.966252
Train Epoch: 8 [3840/6000]      Loss: 0.992912
Train Epoch: 8, Accuracy: 78.300000%

Epoch: 9
Train Epoch: 9 [0/6000] Loss: 0.840200
Train Epoch: 9 [3840/6000]      Loss: 0.935559
Train Epoch: 9, Accuracy: 79.300000%
Accuracy of the network on the test images: 79 %
```

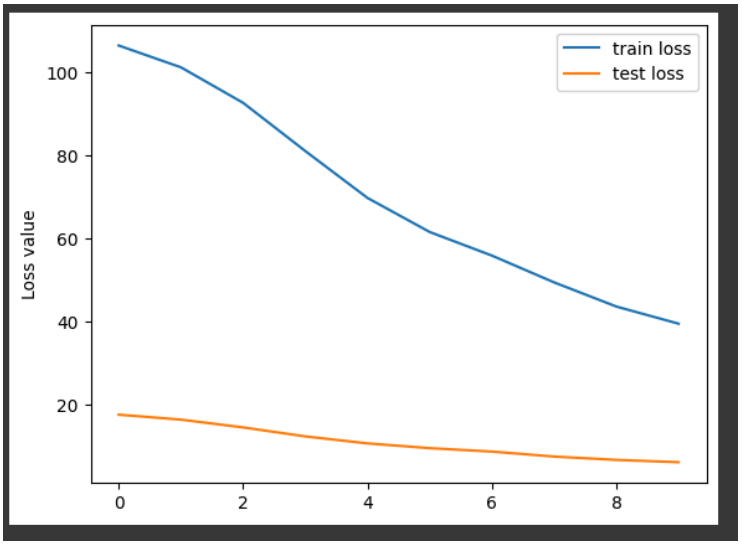
```
Epoch: 0
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: Runt
  self.pid = os.fork()
Train Epoch: 0 [0/6000] Loss: 2.289823
Train Epoch: 0 [3840/6000]      Loss: 2.258880
Train Epoch: 0, Accuracy: 37.300000%

Epoch: 1
Train Epoch: 1 [0/6000] Loss: 2.210599
Train Epoch: 1 [3840/6000]      Loss: 2.159476
Train Epoch: 1, Accuracy: 50.200000%

Epoch: 2
Train Epoch: 2 [0/6000] Loss: 2.069385
Train Epoch: 2 [3840/6000]      Loss: 1.875996
Train Epoch: 2, Accuracy: 56.400000%

Epoch: 3
Train Epoch: 3 [0/6000] Loss: 1.798005
Train Epoch: 3 [3840/6000]      Loss: 1.709092
Train Epoch: 3, Accuracy: 61.800000%

Epoch: 4
Train Epoch: 4 [0/6000] Loss: 1.461067
Train Epoch: 4 [3840/6000]      Loss: 1.260377
Train Epoch: 4, Accuracy: 64.600000%

Epoch: 5
Train Epoch: 5 [0/6000] Loss: 1.427658
Train Epoch: 5 [3840/6000]      Loss: 1.227893
Train Epoch: 5, Accuracy: 67.400000%

Epoch: 6
Train Epoch: 6 [0/6000] Loss: 1.257204
Train Epoch: 6 [3840/6000]      Loss: 1.129881
Train Epoch: 6, Accuracy: 68.800000%

Epoch: 7
Train Epoch: 7 [0/6000] Loss: 0.993114
Train Epoch: 7 [3840/6000]      Loss: 1.153321
Train Epoch: 7, Accuracy: 69.700000%

Epoch: 8
Train Epoch: 8 [0/6000] Loss: 1.137432
Train Epoch: 8 [3840/6000]      Loss: 1.063296
Train Epoch: 8, Accuracy: 70.300000%

Epoch: 9
Train Epoch: 9 [0/6000] Loss: 0.936695
Train Epoch: 9 [3840/6000]      Loss: 0.990733
Train Epoch: 9, Accuracy: 71.000000%
Accuracy of the network on the test images: 71 %
```

The optimiser function seems to keep the weight in check creating noticeable but not extreme changes within loss and accuracy. The accuracy and overall loss does not deviate too far from the original metric.

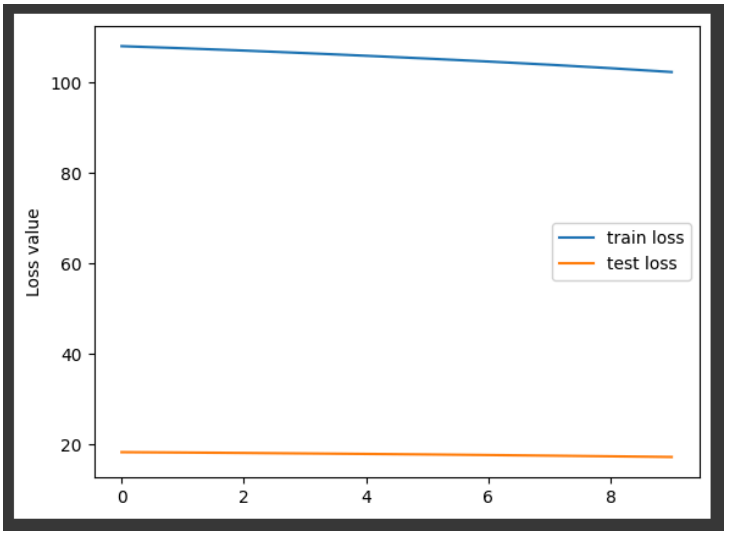
Momentum = 0.9	Momentum = 0.1

```
print("Predicted labels:",samples_pred)
```

```
Original labels: tensor([0, 2, 3, 4, 5, 6, 7, 8])
Predicted labels: tensor([0, 1, 3, 4, 5, 6, 7, 8])
```



```
Original labels: tensor([0, 2, 3, 4, 5, 6, 7, 8])
Predicted labels: tensor([0, 1, 2, 9, 5, 0, 7, 9])
```

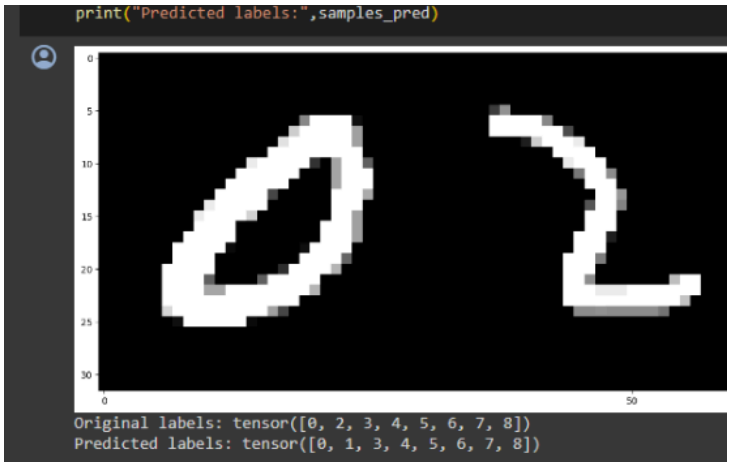
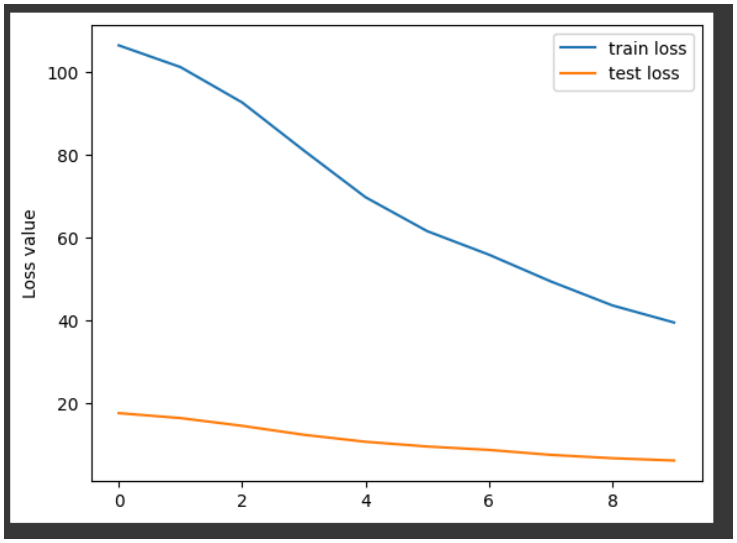
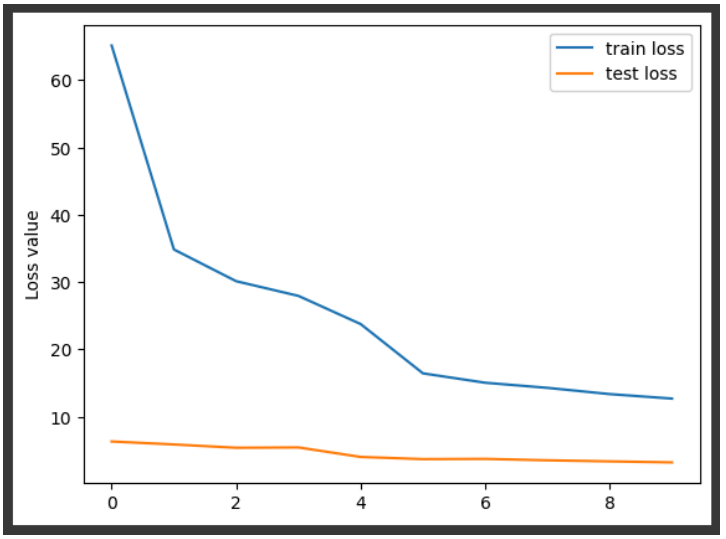


```
Epoch: 0
Train Epoch: 0 [0/6000] Loss: 2.302234
Train Epoch: 0 [3840/6000]      Loss: 2.262031
Train Epoch: 0, Accuracy: 45.200000%

Epoch: 1
Train Epoch: 1 [0/6000] Loss: 2.222993
Train Epoch: 1 [3840/6000]      Loss: 2.147308
Train Epoch: 1, Accuracy: 50.600000%

Epoch: 2
Train Epoch: 2 [0/6000] Loss: 2.019761
Train Epoch: 2 [3840/6000]      Loss: 1.943141
Train Epoch: 2, Accuracy: 56.500000%

Epoch: 3
Train Epoch: 3 [0/6000] Loss: 1.912028
Train Epoch: 3 [3840/6000]      Loss: 1.771532
Train Epoch: 3, Accuracy: 63.000000%

Epoch: 4
Train Epoch: 4 [0/6000] Loss: 1.614347
Train Epoch: 4 [3840/6000]      Loss: 1.485608
Train Epoch: 4, Accuracy: 68.500000%

Epoch: 5
Train Epoch: 5 [0/6000] Loss: 1.539598
Train Epoch: 5 [3840/6000]      Loss: 1.258721
Train Epoch: 5, Accuracy: 70.000000%

Epoch: 6
Train Epoch: 6 [0/6000] Loss: 1.129210
Train Epoch: 6 [3840/6000]      Loss: 1.161818
Train Epoch: 6, Accuracy: 72.100000%

Epoch: 7
Train Epoch: 7 [0/6000] Loss: 1.249073
Train Epoch: 7 [3840/6000]      Loss: 0.886604
Train Epoch: 7, Accuracy: 76.200000%

Epoch: 8
Train Epoch: 8 [0/6000] Loss: 0.966252
Train Epoch: 8 [3840/6000]      Loss: 0.992912
Train Epoch: 8, Accuracy: 78.300000%

Epoch: 9
Train Epoch: 9 [0/6000] Loss: 0.840200
Train Epoch: 9 [3840/6000]      Loss: 0.935559
Train Epoch: 9, Accuracy: 79.300000%
Accuracy of the network on the test images: 79 %
```

```
Epoch: 0
Train Epoch: 0 [0/6000] Loss: 2.306960
Train Epoch: 0 [3840/6000]      Loss: 2.300794
Train Epoch: 0, Accuracy: 16.300000%

Epoch: 1
Train Epoch: 1 [0/6000] Loss: 2.301792
Train Epoch: 1 [3840/6000]      Loss: 2.279794
Train Epoch: 1, Accuracy: 20.700000%

Epoch: 2
Train Epoch: 2 [0/6000] Loss: 2.277868
Train Epoch: 2 [3840/6000]      Loss: 2.271982
Train Epoch: 2, Accuracy: 25.600000%

Epoch: 3
Train Epoch: 3 [0/6000] Loss: 2.259593
Train Epoch: 3 [3840/6000]      Loss: 2.265294
Train Epoch: 3, Accuracy: 30.500000%

Epoch: 4
Train Epoch: 4 [0/6000] Loss: 2.273097
Train Epoch: 4 [3840/6000]      Loss: 2.251313
Train Epoch: 4, Accuracy: 36.300000%

Epoch: 5
Train Epoch: 5 [0/6000] Loss: 2.254121
Train Epoch: 5 [3840/6000]      Loss: 2.245495
Train Epoch: 5, Accuracy: 39.900000%

Epoch: 6
Train Epoch: 6 [0/6000] Loss: 2.240481
Train Epoch: 6 [3840/6000]      Loss: 2.220137
Train Epoch: 6, Accuracy: 42.200000%

Epoch: 7
Train Epoch: 7 [0/6000] Loss: 2.229140
Train Epoch: 7 [3840/6000]      Loss: 2.206033
Train Epoch: 7, Accuracy: 45.100000%

Epoch: 8
Train Epoch: 8 [0/6000] Loss: 2.225345
Train Epoch: 8 [3840/6000]      Loss: 2.209337
Train Epoch: 8, Accuracy: 47.200000%

Epoch: 9
Train Epoch: 9 [0/6000] Loss: 2.187764
Train Epoch: 9 [3840/6000]      Loss: 2.185706
Train Epoch: 9, Accuracy: 47.900000%
Accuracy of the network on the test images: 47 %
```

Momentum is an important parameter to maintain steady loss towards convergence. It compensates for any underfitting that may occur thus better guiding the loss rate towards a more accurate value. Too much or too little momentum and the loss function over or under

fits. In this case the overfitting from a lack of momentum has caused almost all predictions to be wrong.

Learning rate = 0.001	Learning rate = 0.1
 	 

```
Epoch: 0
Train Epoch: 0 [0/6000] Loss: 2.302234
Train Epoch: 0 [3840/6000]      Loss: 2.262031
Train Epoch: 0, Accuracy: 45.200000%

Epoch: 1
Train Epoch: 1 [0/6000] Loss: 2.222993
Train Epoch: 1 [3840/6000]      Loss: 2.147308
Train Epoch: 1, Accuracy: 50.600000%

Epoch: 2
Train Epoch: 2 [0/6000] Loss: 2.019761
Train Epoch: 2 [3840/6000]      Loss: 1.943141
Train Epoch: 2, Accuracy: 56.500000%

Epoch: 3
Train Epoch: 3 [0/6000] Loss: 1.912028
Train Epoch: 3 [3840/6000]      Loss: 1.771532
Train Epoch: 3, Accuracy: 63.000000%

Epoch: 4
Train Epoch: 4 [0/6000] Loss: 1.614347
Train Epoch: 4 [3840/6000]      Loss: 1.485608
Train Epoch: 4, Accuracy: 68.500000%

Epoch: 5
Train Epoch: 5 [0/6000] Loss: 1.539598
Train Epoch: 5 [3840/6000]      Loss: 1.258721
Train Epoch: 5, Accuracy: 70.000000%

Epoch: 6
Train Epoch: 6 [0/6000] Loss: 1.129210
Train Epoch: 6 [3840/6000]      Loss: 1.161818
Train Epoch: 6, Accuracy: 72.100000%

Epoch: 7
Train Epoch: 7 [0/6000] Loss: 1.249073
Train Epoch: 7 [3840/6000]      Loss: 0.886604
Train Epoch: 7, Accuracy: 76.200000%

Epoch: 8
Train Epoch: 8 [0/6000] Loss: 0.966252
Train Epoch: 8 [3840/6000]      Loss: 0.992912
Train Epoch: 8, Accuracy: 78.300000%

Epoch: 9
Train Epoch: 9 [0/6000] Loss: 0.840200
Train Epoch: 9 [3840/6000]      Loss: 0.935559
Train Epoch: 9, Accuracy: 79.300000%
Accuracy of the network on the test images: 79 %
```

```
Epoch: 0
Train Epoch: 0 [0/6000] Loss: 2.299817
Train Epoch: 0 [3840/6000]      Loss: 0.973020
Train Epoch: 0, Accuracy: 70.900000%

Epoch: 1
Train Epoch: 1 [0/6000] Loss: 0.698439
Train Epoch: 1 [3840/6000]      Loss: 0.673693
Train Epoch: 1, Accuracy: 73.000000%

Epoch: 2
Train Epoch: 2 [0/6000] Loss: 0.793851
Train Epoch: 2 [3840/6000]      Loss: 0.410856
Train Epoch: 2, Accuracy: 74.700000%

Epoch: 3
Train Epoch: 3 [0/6000] Loss: 0.471008
Train Epoch: 3 [3840/6000]      Loss: 0.603023
Train Epoch: 3, Accuracy: 74.900000%

Epoch: 4
Train Epoch: 4 [0/6000] Loss: 0.574676
Train Epoch: 4 [3840/6000]      Loss: 0.591790
Train Epoch: 4, Accuracy: 81.900000%

Epoch: 5
Train Epoch: 5 [0/6000] Loss: 0.379718
Train Epoch: 5 [3840/6000]      Loss: 0.387547
Train Epoch: 5, Accuracy: 82.000000%

Epoch: 6
Train Epoch: 6 [0/6000] Loss: 0.400086
Train Epoch: 6 [3840/6000]      Loss: 0.257591
Train Epoch: 6, Accuracy: 83.200000%

Epoch: 7
Train Epoch: 7 [0/6000] Loss: 0.262841
Train Epoch: 7 [3840/6000]      Loss: 0.291743
Train Epoch: 7, Accuracy: 82.900000%

Epoch: 8
Train Epoch: 8 [0/6000] Loss: 0.232796
Train Epoch: 8 [3840/6000]      Loss: 0.321769
Train Epoch: 8, Accuracy: 83.100000%

Epoch: 9
Train Epoch: 9 [0/6000] Loss: 0.260099
Train Epoch: 9 [3840/6000]      Loss: 0.235635
Train Epoch: 9, Accuracy: 83.900000%
Accuracy of the network on the test images: 83 %
```

The learning rate appeared to have the highest impact on the predictive results outputting a perfect score. Although this may seem ideal at first the graph that depicts the loss is jagged and not a consistent curve. Furthermore this means that it is reaching convergence at a more rapid rate so eventually there will be an undershoot of the loss if more epochs were added providing a false sense of accuracy.The Neural network could benefit from a higher

learning rate to achieve accuracy much quicker while preventing overfitting; thus saving on time and resource costs. This makes the CNN model more at risk to the effectiveness of FGSM attacks due to the perturbation images affecting the accuracy more due to a volatile loss rate.

Having an understanding of how changes to any function can have an immense impact on neural network accuracy; A fast gradient sign method will be used to conduct an adversarial attack on the CNN.

The Adversarial attack

Fast gradient sign method (FGSM) is one of the most reliable and effective adversarial attacks due to its simplicity and versatility. FGSM can be catered to different levels of knowledge about the NN it targets, for NN that have no classes, untargeted FGSM can be used. For this CNN it will be assumed that the attacker has complete knowledge of how the CNN discussed in this report functions; therefore FGSM will be used to target classes. FGSM perturbates the images using the following formula:

$$adv\_x = x + \epsilon * sign(grad)$$

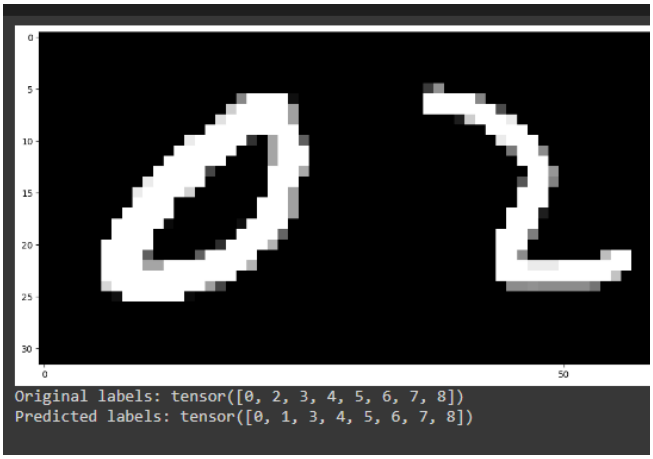
Epsilon is the primary additive that changes the initial loss gradient so it collides with a different class; with x being the initial image gradient. The sign input is the gradient loss of a specific class image. Shown in figure 10, is the FGSM class where the loss function criterion is used to gather the gradients x for the targets (classes). The perturbation is then added to create the adversarial image.

```
[32] base_eps = 1/255
     def fgsm(x,targets,net,epsilon):
         # Collect the element-wise sign of the data gradient
         x.requires_grad = True
         outputs= net(x)
         net.zero_grad()
         loss = criterion(outputs, targets)
         loss.backward()
         data_grad = x.grad.data
         sign_data_grad = data_grad.sign()

         # compute the perturbation
         perturbation = epsilon*sign_data_grad

         # add the perturbation to the original image
         perturbed_image = x + perturbation
         outputs= net(perturbed_image)
         return perturbed_image,perturbation,outputs
```

Figure 10

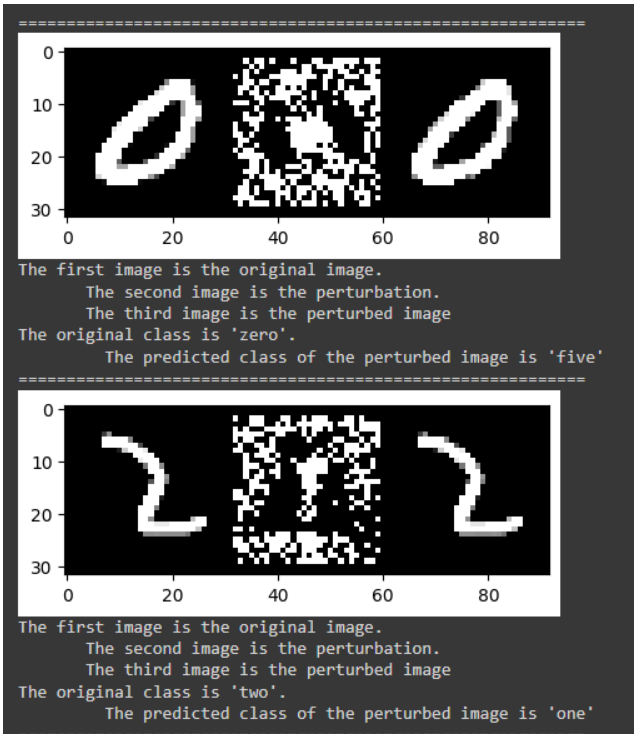
Before Adversarial attack:

The initial prediction with an accuracy of 79% is shown below

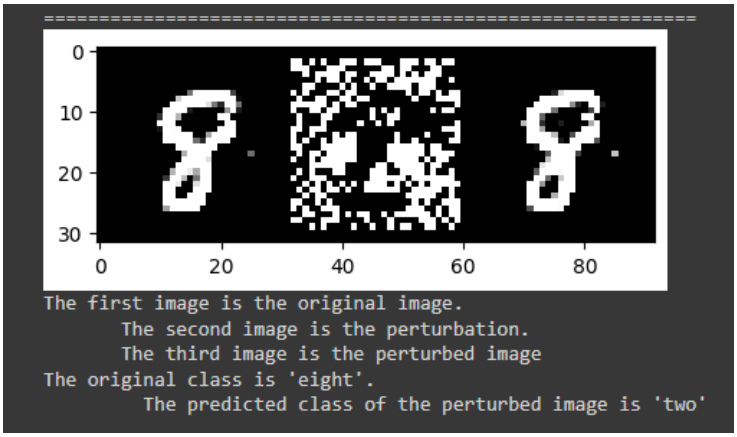


After Adversarial attack:

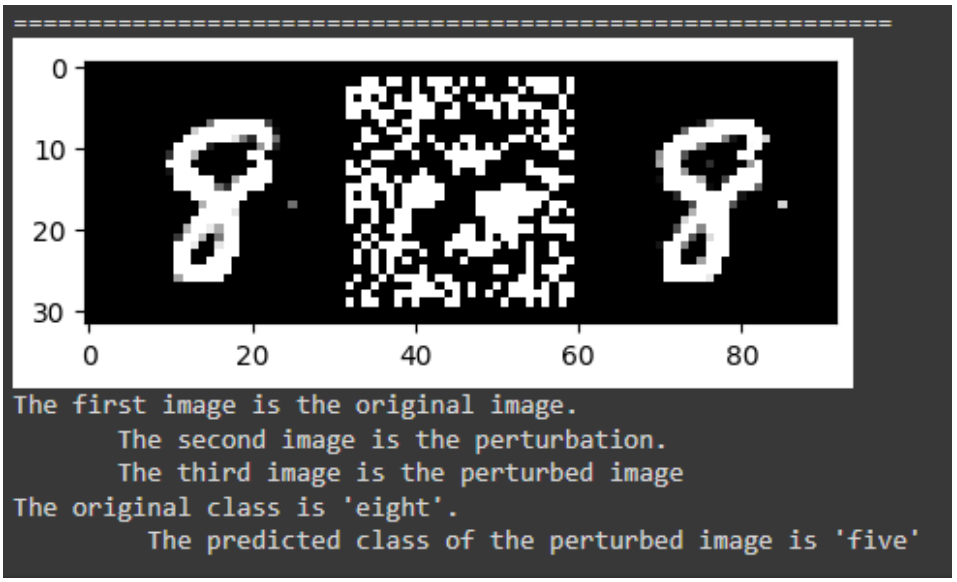
The perturbation images that CNN detected were classified wrong. The perturbation adds noise as shown in the second image which shifts the gradient of the data point to sit where the class zero data points sit.



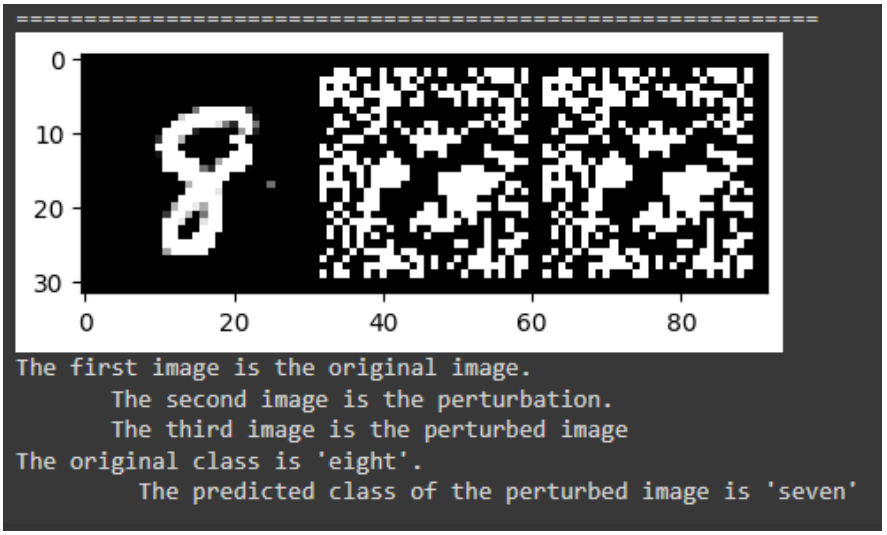
Changing the epsilon from 1/255 to 10/255 to 2000/255:



Epsilon 10/255



Epsilon 2000/255



As depicted in the above figures, small changes can have a large effect on whether the network classifies the numbers correctly or not. As shown in section 6 of the model, the effect metric changes have on the final output can rapidly change how the hidden layer computes and processes the input gradients. The interaction between loss and the learning rate shows how small changes to the loss gradients cause extreme results in accuracy and perturbations. Due to this interaction the effectiveness of FGSM has been quantified and proves the fragility of neural networks. Examples of FGSM can be seen within society and its effect on adopted neural networks. For example:

- This is especially a concern as self-driving vehicles rely on similar CNN models to detect their surroundings. If an object/classification of a human data point is perturbation the CNN model might miss-classify the human object for a road; consequently becoming dangerous towards public safety

3. Metrics and results

MERGED WITH SECTION 2 REASONABLE SOLUTION ON THE PROJECT!

4. Conclusion

The advancements in Artificial intelligence will pave the way for rapid adoptions throughout all aspects of business and personal life. As a result new methods of attack that aim to compromise the integrity of the NN will become more prevalent putting at risk the privacy and safety of individuals. These will come in the form of adversarial attacks such as FGSM that will be used to target critical CNNs, that may in the future; play a critical role in Autonomous vehicles, banking, education and other services. Therefore it is important to understand the parameters used in the various loss, optimisation and gradient functions to gain a better understanding of how adversarial attacks impact CNNs.

References

- Lab 7 - Adversarial_Attack_2024 notebook(Data set, code, result figures, CNN and adversarial attack)
- Linear — PyTorch 1.8.0 documentation. (n.d.). Pytorch.org. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html
- (2024, January). Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy [Review of Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy]. Datacamp.com; Kurtis pykes. https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning
- Multi-layer Perceptron a Supervised Neural Network Model using Sklearn. (2023, October 12). GeeksforGeeks. https://www.geeksforgeeks.org/multi-layer-perceptron-a-supervised-neural-network-model-using-sklearn/
- Mandal, M. (2021, May 1). CNN for Deep Learning | Convolutional Neural Networks (CNN). Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/
- Jason Brownlee. (2019, January 24). Understand the Impact of Learning Rate on Neural Network Performance. Machine Learning Mastery. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

- https://www.facebook.com/jason.brownlee.39. (2016, March 22). Gradient Descent For Machine Learning. Machine Learning Mastery. https://machinelearningmastery.com/gradient-descent-for-machine-learning/
- Brownlee, J. (2021, January 31). Difference Between Backpropagation and Stochastic Gradient Descent. Machine Learning Mastery. https://machinelearningmastery.com/difference-between-backpropagation-and-stochastic-gradient-descent/
- Gomes, J. (2018, January 17). Adversarial Attacks and Defences for Convolutional Neural Networks. Onfido Tech. https://medium.com/onfido-tech/adversarial-attacks-and-defences-for-convolutional-neural-networks-66915ece52e7
- Sciforce. (2022, September 7). Adversarial Attacks Explained (And How to Defend ML Models Against Them). Sciforce. https://medium.com/sciforce/adversarial-attacks-explained-and-how-to-defend-ml-models-against-them-d76f7d013b18
- Sen, J., Sen, A., & Chatterjee, A. (2023). Adversarial Attacks on Image Classification Models: Analysis and Defense. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.2312.16880
- Campagne, J.-E. (2020). Adversarial training applied to Convolutional Neural Network for photometric redshift predictions. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.2002.10154
- Python, R. (n.d.). Stochastic Gradient Descent Algorithm With Python and NumPy – Real Python. Realpython.com. https://realpython.com/gradient-descent-algorithm-python/#:~:text=Stochastic%20gradient%20descent%20is%20an
- Roy, R. (2019, February 15). ML | Stochastic Gradient Descent (SGD). GeeksforGeeks. https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/