Faculty of Natural and Mathematical Sciences
Department of Informatics

Bush House, King's College London, Strand Campus, 30 Aldwych, London WC2B 4BG
Telephone 020 7848 2145
Fax 020 7848 2851

7CCSMPRJ

Individual Project Submission 2020/21

Project Title: Adversarial robustness of deep learning using dropout

Table of contents

List of figures

List of tables

Nomenclature

$\boldsymbol{b}^l$	Vector of biases between layers $l-1$ and l	x'	Adversarial example for x
D	Distance function	y	True label of input x
f	Activation function	$\boldsymbol{z}^l$	Activation vector
F	Machine learning model e.g., Neural network classifier	$Z(x)_i$	Logit value for input x for class i
J	Loss function value		**Greek Symbols**
J_{adv}	Adversarial risk	α	Learning rate
L_p	L_p norm	ε	Perturbation budget
p	Dropout rate	θ	Parameters of F
S	Input space	μ	Adversarial severity
t	Target label for x'	ρ	Adversarial robustness
$\boldsymbol{W}^l$	Matrix of weights between layers $l-1$ and l	Φ	Adversarial frequency
x	Original input	$\|\cdot\|_p$	L_p norm (alternative notation)

Note that in certain parts of the present work this nomenclature is slightly abused, but in such cases the interpretation of symbols is explained in the text.

1. Introduction

1.1. Neural networks and their applications

As the field machine learning has emerged over the past few years, neural networks (NN), a specific subset of machine learning models, have been at the forefront of its development.

An NN is defined as a collection of interconnected nodes. Signals, or activations, are transmitted from one node to another through connections, each of which has an associated weight which impacts the signal's strength. The nodes are organised into layers, with an input layer, an output layer, and most often at least one hidden layer in between them. At the input layer, the feature vector of an input is inputted, each node corresponding to a feature, and at the output layer, the final prediction of the model for the input is outputted. NNs can generally be divided into two categories: feedforward neural networks (FFNN) and recurrent neural networks (RNN). In FFNNs, signals are only sent forward through the layers, meaning away from the input layer and towards the output layer. In RNNs, signals can be sent backwards as well as forwards.

A fully connected NN, shown in figure 1 [1], is an NN where every node in a layer is connected to every node in the next layer. At each node, every incoming signal is multiplied by the weight of the connection it travelled along, and the sum of all these products and a bias term is taken. This scalar is then passed through an activation function, such as the sigmoid or ReLU function, to produce the output signal of the node. For a layer $\boldsymbol{l}$, the processing of inputs to outputs is described by equation 1:

$$\boldsymbol{z}^{(l)} = f\left(\boldsymbol{W}^{(l)}\boldsymbol{z}^{(l-1)} + \boldsymbol{b}^{(l)}\right) \tag{1}$$

A 3-layer NN, also seen in figure 1 [1], is an NN composed of a hidden layer as well as the input and output layers, with the hidden layer placed between the input and output layers. According to the universal approximation theorem, this neural network can implement any continuous function and therefore can be trained to solve any classification task. However, the more complex these tasks become, the more nodes and parameters it requires to perform well, and as a result it becomes a computationally impractical solution.

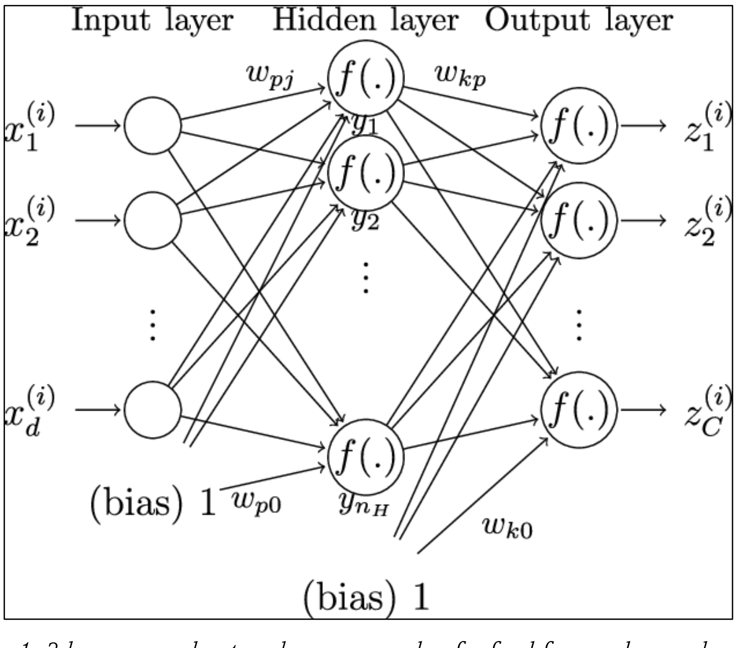


Figure 1: 3-layer neural network, an example of a feed-forward neural network

Deep neural networks (DNN) are a more viable alternative. A DNN is an NN with more than one hidden layer, and any learning algorithm used to train a DNN falls under the category of deep learning. The purpose of having multiple layers is for the network to learn the complex functions needed to perform well in these complex tasks with a smaller increase in the number of parameters than the 3-layer network would require. The popularity of and reliance on deep learning can be attributed to its success in applications such as computer vision [2]–[5], speech recognition [6]–[9], and natural language processing [10], [11], as well as in niche applications like video-game playing [12], big data analytics [13] and medical imaging [14].

Convolutional neural networks (CNN) are a type of DNN, used predominantly in image classification. CNNs include convolutional layers. These have kernels, instead of nodes and weighted connections, which are applied to various combinations of pixels in the input channels, as feature detectors. This is done to save memory space by weight sharing; by using kernels instead of connections, multiple connections which detect the same feature in different positions in the image, and thus would have the same weights, can be consolidated into a single value [15]. Pooling layers are also included in CNNs and serve the same purpose. Figure 2 [16] shows the architecture of a typical CNN.

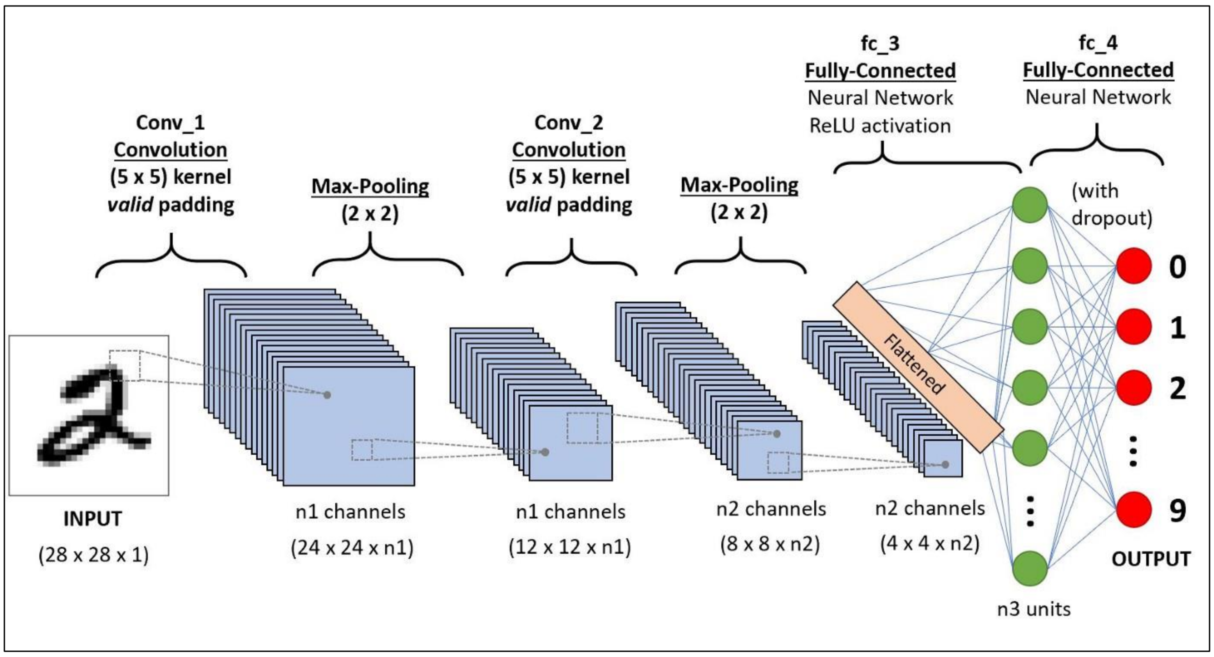


Figure 2: Example architecture of a convolutional neural network

For all NNs used in classification tasks, the output vector of the output layer prior to the application of the activation function is called the logits vector. Each element in the vector is considered an unnormalized prediction for its corresponding class. To normalise these predictions, the SoftMax activation function is usually applied. The resulting vector is often interpreted as the probability of each class. A common loss function used to train NNs is cross-entropy loss.

1.2. Dropout

A challenge facing all machine learning models is preventing overfitting to training data to ensure good generalisation. Methods used to achieve this are called regularisation methods. DNNs are particularly prone to overfitting as they have many parameters and use non-linear functions, so they often require regularisation.

One regularisation technique for DNNs which has been shown to improve generalisation is dropout [17]. In dropout, in each training iteration, various nodes and their incoming and outgoing connections are dropped from the network, and the training iteration (both in the forward pass step and backpropagation step) is only applied to the 'thinned' network that remains [18]. The nodes which are dropped out are chosen at random; each node undergoes an independent Bernoulli trial with success probability p to determine if it's dropped out, with p referred to as the dropout rate.

Dropout is never applied to the output layer of an NN as all the nodes in this layer are necessary for the classification of a sample. Therefore, to ensure that the magnitude of the expected output at the output layer of the network when dropout is applied during training is approximately equal to the expected output of the final layer at test time, the activations of the nodes which are not dropped are multiplied by $\frac{1}{1-p}$. As a result of this, the expected output of each layer where dropout is applied remains equal to the expected output of the layer had dropout not been applied.

Figure 3 [19] shows a visual representation of dropout. Essentially, through dropout, an exponential number of thinned networks are trained, and are then combined at test time to find an approximate of their average, in a bid to improve generalisation. Other types of regularisation include L1 and L2 regularisation and data augmentation.

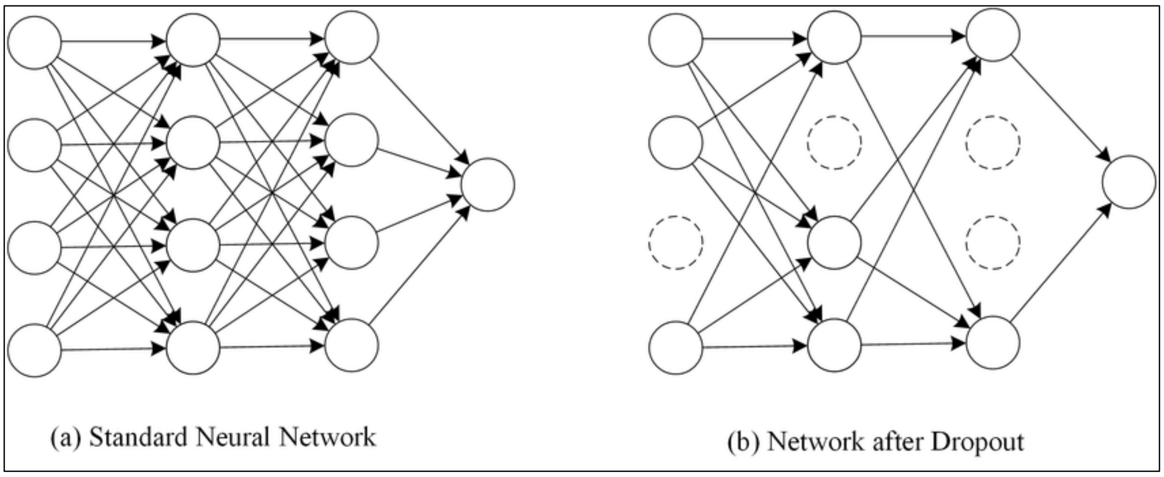


Figure 3: A neural network (a) without dropout and (b) with dropout

1.3. Ensemble classifiers

The goal of a learning algorithm is to find the hypothesis that will produce the best predictions for a given task. Nevertheless, it's possible that no single hypothesis will perform well individually, or that multiple hypotheses will perform equally well overall but differently in subsections of the task.

Ensemble classifiers try to solve these problems. An ensemble classifier, such as the one shown in figure 4 [20], is a finite set of individual classifiers whose outputs for a given input are combined using a combination function to produce a single final output for that input. Ideally, each classifier learns a different hypothesis, and when these hypotheses are combined into one, the weak area of each individual classifier is compensated for by the strength of the other classifiers in that area. Since the individual classifiers should be distinct, the prime challenge when creating an ensemble classifier is ensuring the diversity of its individual classifiers. Random forest [21], an ensemble of decision tree classifiers, is a widely used ensemble classifier.

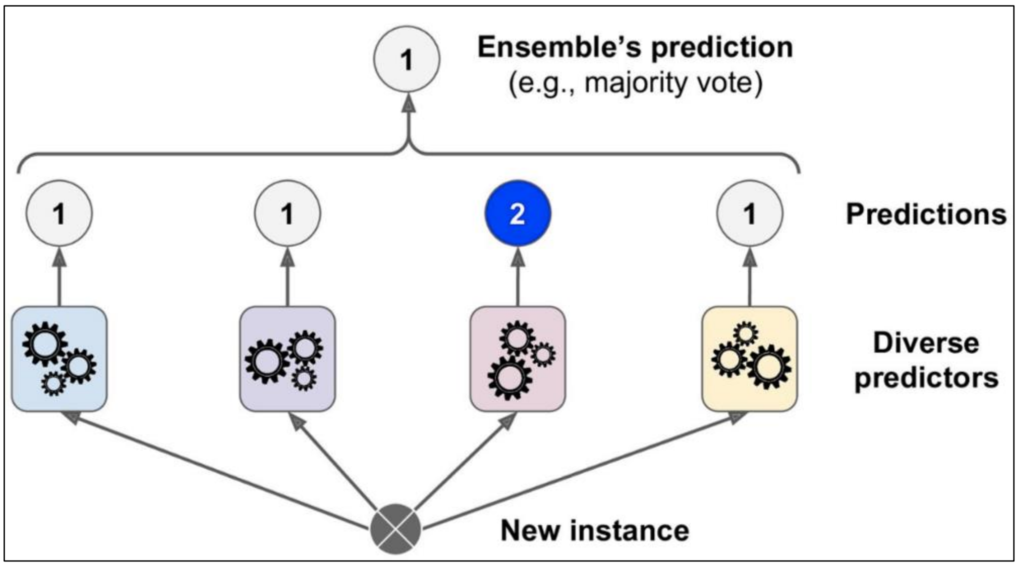
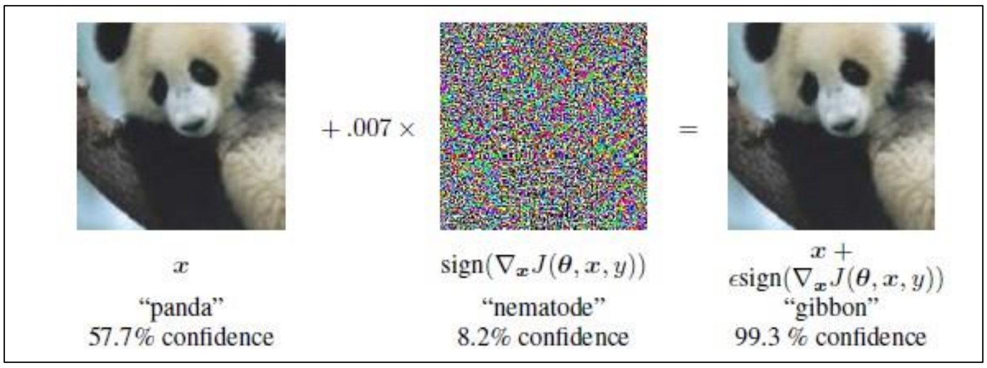


Figure 4: Visual representation of the structure of an ensemble classifier

1.4. Adversarial examples

Another challenge facing machine learning models is the phenomenon of adversarial examples, which was discovered by Szegedy et al [22] in 2014. These are defined as inputs which are incorrectly classified by a machine learning model despite not being perceptibly different from inputs that are correctly classified by the same machine learning model. An example [23] can be seen below in figure 5. Despite the original and modified images being imperceivably different, the classifier outputs an incorrect prediction for the modified image. As a result of this phenomenon, machine learning models are vulnerable to adversarial attacks, which are when adversarial examples are used to induce machine learning models to produce incorrect predictions to cause undesirable effects. The existence of adversarial examples has led industrial users of machine learning models to be cautious when applying them in safety-critical applications. Since the phenomenon's discovery, research has focused on finding its causes [22]–[29], developing methods for generating adversarial examples, and creating strategies for defending against adversarial attacks [30]–[34].



$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 5: An adversarial example, formed by adding specific noise to an original input

1.5. Project aim

Although dropout is mainly used in the training phase of DNNs, it could technically also be used in the inference phase, by dropping nodes and their corresponding weights when an input is passed in for inference. As dropout is random, passing a given input multiple times into a network using dropout during inference will produce multiple, possibly different, predictions, resembling an ensemble. The goal of the present work is to determine whether using dropout in this way, which we call 'test-time dropout', can be used as a defence strategy to increase a DNN's resistance to adversarial attacks.

1.6. Objectives

To achieve this aim, the following objectives, best thought of as sub-tasks, must be met:

1. **Conduct review of existing literature relevant to the project aim.** This review will include the following topics:
    - Threat models of adversarial attacks
    - Adversarial example generation methods
    - Existing defence strategies against adversarial attacks, especially those using randomisation-based schemes
    - Metrics used for evaluating the adversarial robustness of a DNN
    - Current usage of test-time dropout

    Coverage of these areas will help the author to gain the fundamental knowledge in the subject area and reveal the subject area's knowledge gaps which the project can aim to fill.
2. **Develop experimental setup for assessing the effect of test-time dropout on adversarial robustness.** Previous work in the area will be helpful for this. The setup should allow for the following features to be variable to facilitate analysis of results with respect to each of them:
    - Adversarial example generation method used
    - Location and type of layers which test-time dropout is applied to
    - Test-time dropout rate
    - Ensemble size
    - Perturbation budget

    The concept of perturbation budget is explained in section 2.2.
3. **Evaluate results parametrically.** The results produced by the experiments will be evaluated to determine the effectiveness of test-time dropout as a defence strategy. Analysing the results with respect to each of the variable parameters listed in objective 2 will indicate how the effectiveness of test-time dropout changes with respect to each of them and thus allow us to learn how to use it most effectively.

1.7. Desired outcome

As the project is agnostic as to whether test-time dropout is found to be effective in improving adversarial robustness, the desired outcome is simply to conduct the experiment in the correct way so that reliable results are found. If test-time dropout is found to be effective, further exploration of it can be done, and if not, further work can focus on other strategies where more promise may lie.

1.8. Report structure

Section 2 provides a review of the literature relevant to the subject, presents relevant background theory, explains why test-time dropout might work and states how the present work intends to fill the uncovered knowledge gaps. The ideas developed there set the context in which the experiments of the project will be done. The details of the experimental setup are given in section 3. In section 4, the experimental results are analysed. Section 5 contains the conclusions drawn from the results and discusses the shortcomings of the present work and how future work might address them.

2. Literature review and background theory

2.1. Threat models

An adversarial attack can be defined by three characteristics: the goal of the attack, the extent of the adversary's knowledge of the victim model, and the capabilities of the adversary [35]. The specific combination of these three characteristics is called the threat model. When assessing the strength of an attack or defence strategy, the threat model should be specified.

An attack can have one of two goals. In a targeted attack, the adversary aims to induce the classifier to classify the input with a specific incorrect label, whereas in a non-targeted attack, the adversary aims to induce the classifier to classify the input with any incorrect label.

With regards to the attacker's knowledge, there are 2 main possible scenarios. If the adversary knows the topology and parameters of the classifier and always has access to them, the attack is known as a white-box attack. If the adversary has no knowledge of the details of the classifier, and all it knows is the final predicted label or the predicted scores for each possible label, it is known as a black-box attack. Outside of these scenarios, there are some generation methods that initially require white-box access, but at the time of querying the adversarial example do not require such access. Such attacks are called semi-white-box or grey-box attacks.

Finally, an adversary can have a range of capabilities. An adversary with unlimited capabilities could theoretically edit the parameters of the victim model, meddle with its training data, or create adversarial examples with such large perturbations from the original input that perceptual similarity is doubtful. Under such circumstances, victim models would be completely exposed, and it would be meaningless to test their robustness against such an adversary, so the capabilities of the adversary must be somewhat curtailed and specified. Common practice is to not allow adversaries access to edit the model parameters or training data, and to restrict the perturbations that the adversary can develop to be within a certain budget to preserve perceptual similarity. This perturbation budget is usually implemented using an L_p norm, a distance metric explained in section 2.2.

2.2. Definitions

Having verbally defined adversarial examples in the introduction and introduced threat models, we now give the mathematical definition of an adversarial example.

Definition 1: In an untargeted attack, x' is an adversarial example for classifier F if:

$$F(x') \neq y \; s.t. D(x, x') \leq \varepsilon \tag{2}$$

Definition 2: In a targeted attack, x' is an adversarial example for classifier F if:

$$F(x') = t \; s.t. D(x, x') \leq \varepsilon \tag{3}$$

In definitions 1 and 2, D denotes a distance function and is used to ensure perceptual similarity between its two arguments, which are the original input x and its corresponding adversarial example x'. It is often implemented using an L_p norm, which for inputs x and x' is defined mathematically as:

$$L_p = \|x - x'\|_p = ((x_1 - x_1')^p + (x_2 - x_2')^p + (x_3 - x_3')^p + \cdots + (x_n - x_n')^p)^{\frac{1}{p}} \tag{4}$$

Typically, a p value of 0, 1, 2, or infinity is used, and in the context of images, these all have intuitive interpretations. L_0 represents the number of pixels which are different between the two images, L_1 is the sum of the differences between all pixels, L_2 is the Euclidean distance between the pixels of the images, and L_∞ represents the maximal difference between pixels in the same location of each image.

There are also basic metrics which must be defined for analysing robustness to adversarial attacks.

Definition 3: Pointwise adversarial robustness is defined as the minimal norm of perturbation required to change F's predicted class at a given x:

$$\rho(x,F) = \min_{\|x-x'\|_p} \|x-x'\|_p \; s.t. F(x') \neq y \tag{5}$$

Definition 4: Global adversarial robustness is defined as the expected value of the pointwise adversarial robustness over F's entire input space S:

$$\rho(F) = E_{x\sim S}\big(\rho(x,F)\big) \tag{6}$$

Definition 5: Pointwise adversarial loss is defined as the maximum loss produced by F for class y over all x' that are within a certain distance of x:

$$J_{adv}(x,F) = \max_{J} J(F,x',y) \; s.t. \|x-x'\|_p \leq \varepsilon \tag{7}$$

Definition 6: Global adversarial loss is defined as the as the expected value of the pointwise adversarial loss over F's entire input space S:

$$J_{adv}(F) = E_{x\sim S}\big(J_{adv}(x,F)\big) \tag{8}$$

An input x is said to be ε-robust to any perturbation value ε less than or equal to $\rho(x,F)$. Similarly, a model F is said to be ε-robust to any perturbation value ε less than or equal to $\rho(F)$. Most works use definitions 3 and 4 to assess the resistance of a machine learning model to adversarial examples as opposed to definitions 5 and 6. Definitions 3 and 4 are discussed further in section 2.5.

2.3. Generation methods

Since the initial discovery of adversarial examples, a great deal of research has gone into developing methods for generating them. To neutralise such threats, defensive strategies against adversarial attacks have also been designed. With each side always trying to better the other, the dynamic has evolved into that of an arms race. Presented first is a review of the generation methods.

2.3.1. White-box methods

As the adversary has access to the victim model's details in a white-box scenario, it can use gradient-based techniques, some of which are iterative/optimisation-based, to generate adversarial examples.

Limited-memory BFGS (L-BFGS) attack

Szegedy et al [22], in their original paper, proposed a generation method for a white-box, targeted attack. They posed the problem as a search for the x' with minimal L_2 perturbation from x, such that x' is classified by the victim model as t. Mathematically, this is written as the following constrained optimisation problem:

$$\underset{x'}{\operatorname{argmin}} \|x-x'\|_2^2, s.t. F(x') = t \neq y \tag{9}$$

The authors then transformed the problem into an unconstrained optimisation problem by adding the loss function into the optimisation problem as a substitute for the constraint:

$$\underset{x'}{\operatorname{argmin}}\, c\|x-x'\|_2^2 + J(F,x',t) \tag{10}$$

The first term, as before, attempts to minimise the L_2 perturbation from x while the second term attempts to minimise the loss of x' relative to the target label t, which maximises the likelihood of x' being classified as t. The constant c is used to balance the trade-off between the competing terms. The authors used the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, a common gradient-based algorithm used in numerical optimisation problems, to solve the problem.

Fast gradient sign method (FGSM)

FGSM is a single-step method that can be used in both targeted and untargeted attacks in a white-box scenario. The following formulae are used:

$$x' = x + \varepsilon \cdot sign\left(\frac{\partial}{\partial x}J(F,x,y)\right)\ (For\ untargeted\ attack) \tag{11}$$

$$x' = x - \varepsilon \cdot sign\left(\frac{\partial}{\partial x}J(F,x,t)\right)\ (For\ targeted\ attack) \tag{12}$$

This method applies gradient ascent/descent but only for a single iteration. In the untargeted setting, the method ascends the gradient of the loss function with respect to x, relative to y, to try to maximise the loss function and thus decrease the probability of the input being classified as y. In the targeted setting, the method descends the gradient to minimise the loss function, relative to t, and thus increase the probability of the input being classified as t. The method was proposed by Goodfellow et al [23] and has the benefit of generating adversarial examples quickly.

Projected gradient descent (PGD) method

PGD method, first proposed by Kurakin et al [36], is an iterative method used for white-box attacks. The update rules used for untargeted and targeted attacks are:

$$x_{i+1} = Clip_{x,\varepsilon}\left(x_i + \alpha\ sign\left(\frac{\partial}{\partial x}J(F,x_i,y)\right)\right)\ with\ initial\ condition\ x_0 = x,\ untargeted\ attack \tag{13}$$

$$x_{i+1} = Clip_{x,\varepsilon}\left(x_i - \alpha\ sign\left(\frac{\partial}{\partial x}J(F,x_i,t)\right)\right)\ with\ initial\ condition\ x_0 = x,\ targeted\ attack \tag{14}$$

The $Clip_{x,\varepsilon}$ function projects its argument, which in this case is x', into the ε-neighbourhood of x. The ε-neighbourhood of x is defined as the set of inputs for which the L_p-norm between the input and x is below or equal to ε. Therefore, the ε-neighbourhood of x is defined by two parameters: the L_p-norm measure to be used and the threshold ε to be used.

Thought of another way, this method uses gradient descent to search for the x' which maximises (for an untargeted attack) or minimises (for a targeted attack) the loss function, with the constraint that the perturbation between x and x' must be less than ε for the specified L_p-norm. These searches could be written mathematically as the following constrained optimisation problems:

$$\underset{x'}{\operatorname{argmax}} J(F,x',y)\,, s.t. \|x - x'\|_p \le \varepsilon\ (For\ untargeted\ attack) \tag{15}$$

$$\underset{x'}{\operatorname{argmin}} J(F,x',t)\,, s.t. \|x - x'\|_p \le \varepsilon\ (For\ targeted\ attack) \tag{16}$$

When written like this, they can be compared with the L-BFGS attack explained earlier, the difference being that the objective and constraint are reversed in this method relative to that one.

Carlini & Wagner (C-W) attack

In response to the development of defences shown to be effective against the L-BFGS method, Carlini & Wagner [37] proposed a more sophisticated version of the method. The authors aimed at solving the same initial constrained optimisation problem, but when converting it into an unconstrained problem, they substituted a different loss function term into the optimisation problem. This term is often referred to as the margin loss [31], and mathematically is written as:

$$\max_{i \neq t}(Z(x')_i - Z(x')_t, 0) \tag{17}$$

$Z(x')_i$ represents the logit for class i. When the logit for the target class t is not the highest logit, $Z(x')_i - Z(x')_t$ will be positive and will thus be the output of the term. When the logit for the target class is the highest logit, $Z(x')_i - Z(x')_t$ is negative, and thus the output of the term is 0.

The unconstrained optimisation problem for this method is as follows:

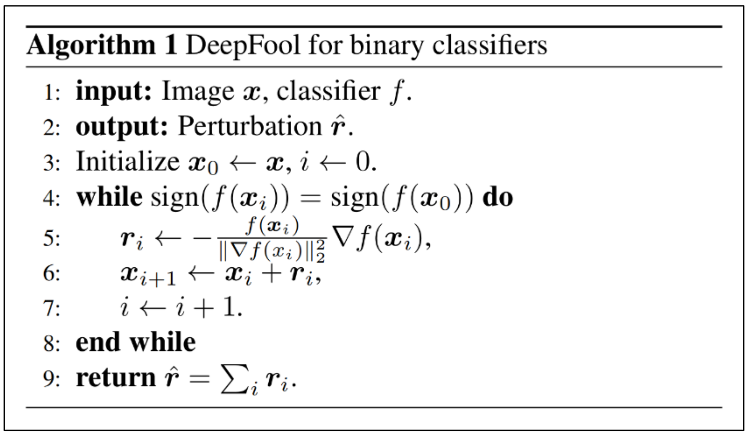
$$\underset{x'}{\operatorname{argmin}} \|x - x'\|_2^2 + c \max_{i \neq t}(Z(x')_i - Z(x')_t, 0) \tag{18}$$

The first term, as with the L-BFGS method, attempts to minimise the L_2 perturbation from x. The second term attempts to ensure that x' is classified as t, but the benefit of this term is that once its goal is achieved, it has a fixed value of 0, and thus the algorithm can focus on minimising the first term only. Again, the constant c is used to control the trade-off between the competing objectives.

The Adam algorithm [38] was used by the authors to solve the problem as it showed superior efficiency, and the authors found their algorithm to be 100% successful against classifiers deploying the defensive strategies which improved their robustness against earlier generation methods [37].

DeepFool

DeepFool [39] is a white-box generation method that is generally used for untargeted attacks but can also be used for targeted attacks if the target label t shares a decision boundary with the true label y. The DeepFool method identifies the decision boundaries around the input, and then finds the transformations to the input which would cause it to cross over the decision boundary and thus be classified incorrectly. Figure 6 [39] contains the pseudocode describing the DeepFool algorithm.



Algorithm 1 DeepFool for binary classifiers

```
1: input: Image $\boldsymbol{x}$, classifier $f$.
2: output: Perturbation $\hat{\boldsymbol{r}}$.
3: Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
4: while $\operatorname{sign}(f(\boldsymbol{x}_i)) = \operatorname{sign}(f(\boldsymbol{x}_0))$ do
5:     $\boldsymbol{r}_i \leftarrow -\frac{f(\boldsymbol{x}_i)}{\|\nabla f(\boldsymbol{x}_i)\|_2^2} \nabla f(\boldsymbol{x}_i)$,
6:     $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$,
7:     $i \leftarrow i + 1$.
8: end while
9: return $\hat{\boldsymbol{r}} = \sum_i \boldsymbol{r}_i$.
```

Figure 6: Pseudocode for the DeepFool algorithm

Other white-box methods

Other white-box generation methods include distribution-based methods, such as Distributionally adversarial attack [40] and AdvGAN [41], and the spatial transformation method [42]. A characteristic of the former two methods is that once the distribution is found, the victim model's details are no longer required when generating adversarial examples, so they are grey-box methods.

2.3.2. Black-box methods

As the adversary has no knowledge of the victim model's details in a black-box scenario, gradient-based optimisation techniques cannot be used when generating adversarial examples, and thus other methods are needed. Black-box methods are generally split into three categories: score-based attacks, transfer-based attacks, and decision-based attacks.

Score-based attacks

Score-based attacks use the logits vector produced by the classifier to generate adversarial examples. The adversary uses the elements of the logits vector to estimate the gradient, which it then uses to create x'. Examples of score-based attacks are Zeroth Order Optimisation method (ZOO) [43], Natural evolution strategy (NES) [44] attack and N-attack [45].

Transfer-based attacks

Transfer attacks rely on the principle of transferability [22], which is the observation that adversarial examples generated for a machine learning model trained for a given task can deceive other machine learning models trained for the same task [22]. Whilst the adversary has no knowledge of the victim model's details in a black-box scenario, transfer-based methods do require either the training data of the victim model or a similar synthetic dataset. Using this dataset, the adversary trains another model for the same task, and uses white-box generation methods on that model to generate adversarial examples, which it then uses to attack the victim model.

Decision-based attacks

Decision-based attacks are for situations where the only information available to the adversary is the final class prediction, meaning the logits aren't even available. The boundary attack [46], the first attack of this kind, used this information by starting at a point which was already adversarial and then used a random walk to minimise the perturbation from x. Whilst it performed well, it suffered from a lack of convergence in some cases, as well as long run times. Attempts to improve the algorithm produced the Query-Efficient Boundary Attack (QEBA) [47] and Hop-Skip-Jump attack [48].

A sub-category of decision-based attacks are noise-based attacks, where an original input x is blended with types of random noise iteratively until an x' is produced which is classified incorrectly by the victim model. An example is the salt-and-pepper noise attack. Whilst it hasn't been formalised in the literature, the Foolbox python package [49] offers an implementation of it.

2.4. Defence strategies

Developing strategies that improve the adversarial robustness of neural networks and neutralise adversarial attacks could be the key to allowing practical applications to take full advantage of the potential shown by deep learning. So far, research in the field has produced strategies that can be classified into 4 broad categories.

2.4.1. Robust optimisation/training

Robust optimisation, the first category, entails training models with the express aim of ensuring adversarial robustness in addition to the standard goal of ensuring good generalisation. Three ways of doing this are adversarial training, provable defences, and ensemble classifiers.

Adversarial training

Adversarial training involves including adversarial examples in a model's training dataset through adapted learning algorithms. Many versions of this defence have been developed, each using different adversarial example generation methods and different implementation frameworks.

In the same work in which they devise the FGSM generation method, Goodfellow et al [23] propose using examples developed with this method for adversarial training. They propose to do this by training the model using the following loss function:

$$\hat{J}(F,x,y) = cJ(F,x,y) + (1-c)J\left(F, x + \varepsilon \cdot sign\left(\frac{\partial}{\partial x}J(F,x,y)\right), y\right) \tag{19}$$

The first term encourages the algorithm to find the solution to minimise the conventional loss of the training example, whilst the second term encourages the algorithm to minimise the loss of the FGSM adversarial example x'corresponding to the original input x for the conventional loss function. The constant c is used to balance these goals. The authors reported [23] that whilst this strategy improved performance against attacks using FGSM, it still left models trained with it vulnerable to attacks which use iterative or optimisation-based generation methods.

In trying to formalise the broad problem of adversarial robustness, Madry et al [50] formulated it as a saddle point optimisation problem, which can be mathematically written as follows:

$$\min_{\theta} E\left(\max_{\|x-x'\|_p} J(\theta, x', y)\right) \tag{20}$$

The inner maximisation problem aims to find an adversarial example for each training dataset sample to the current iteration of the network, i.e., with parameters $\boldsymbol{\theta}$, by maximising the adversarial risk. The outer minimisation problem aims to find parameters of the NN, $\boldsymbol{\theta}$, that minimise the expected adversarial risk of these adversarial examples. Notice that the inner maximisation problem is a possible aim of an adversarial attack, whilst the outer minimisation problem is a possible aim for a model trying to be adversarially robust. Therefore, solving this saddle optimisation problem is equivalent to performing adversarial training. The generation technique used in the inner maximisation problem is the PGD technique, as the authors found it to be representative of most first-order attacks and thus felt that training the model with this method would provide robustness to all other first-order attacks. The L_p-norm used was the L_∞-norm. Whilst the experiments performed by the authors [50] using PGD adversarial training proved that it is effective, they also found that it is computationally expensive to implement and thus is often not a realistic solution.

Other adversarial training strategies include adversarial logit pairing and ensemble adversarial training. In adversarial logit training [51], the following loss function with a novel second term is used to train the model:

$$\hat{J}(F,x,y) = J(F,x,y) + cJ_{ALP}\big(F, F(x), F(x')\big) \tag{21}$$

This second term, J_{ALP}, encourages the algorithm to minimise the difference between the logits outputted by the network for an input x and its corresponding adversarial example x'. Any loss function that encourages the two sets of logits to be similar can be used for J_{ALP}. In Kannan et al's [51] version of adversarial logit training, the PGD method is used to generate adversarial examples. Ensemble adversarial training [52] utilises the transferability property of adversarial examples by inserting adversarial examples developed on other models into the training dataset of a model.

Provable defences

A shortcoming of adversarial training is that the model is only trained to be robust against adversarial examples generated by the specific generation technique used in the learning algorithm. Even if the technique used is stronger than all existing techniques, stronger techniques may emerge in the future which will be able to find adversarial examples with smaller perturbations, and adversarially trained models would not be robust to attacks using this technique. Therefore, provable defences aim to train models to maximise the true adversarial robustness of neural networks as opposed to maximising their robustness to specific attacks.

Hein and Andruschenko [53] derived an inequality expression for the objective adversarial robustness of a model at a given point in the input space. The authors then developed an adapted training loss function to be applied to the model to maximise this lower bound for a range of input space points and thus improve adversarial robustness. Similarly, Raghunathan et al [54] derived an inequality expression for the objective adversarial margin loss of a model at a given point in the input space and developed a training method to minimise this upper bound. Both works addressed neural networks with only a single hidden layer, but further works [55], [56] have addressed how to scale up the strategies discussed in them to deeper neural networks.

Ensemble classifiers

The point of using an ensemble of models in a classifier is to minimise the adversarial subspace of the final classifier. As there are multiple models in the classifier, to fool the final classifier, the adversary must devise a perturbation which falls within the shared adversarial subspace of at least one combination of half of these models. This means that the adversarial subspace of the final classifier is the subspace within the input space where at least half of the models within the ensemble are adversarial, i.e., the space where the adversarial subspaces of at least half of the models overlap. The lesser this overlap is, the smaller the total adversarial subspace of the final classifier is, and the smaller the adversarial subspace is, the harder it is for a given adversary to find it. As a result of this, the main challenge when using an ensemble for adversarial robustness is to ensure that the adversarial subspaces of the models are diverse. Figure 7 [57] shows this visually.

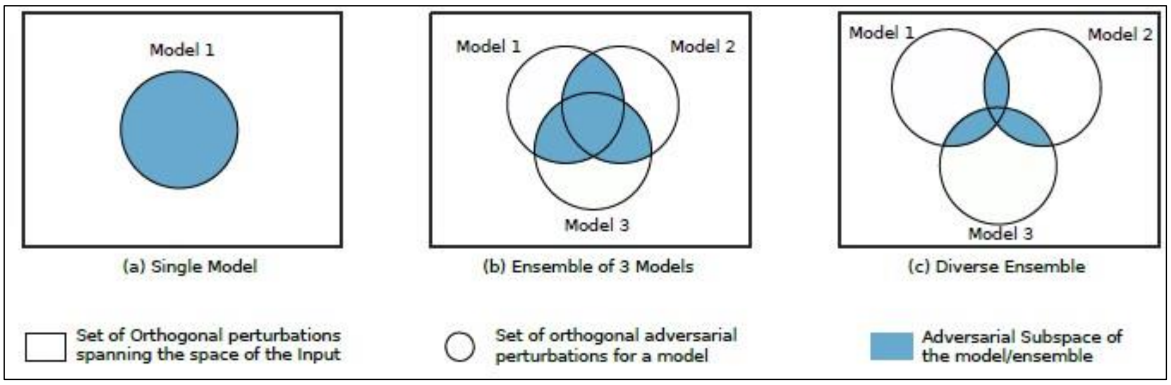

Model 1
(a) Single Model
Model 1
Model 2
Model 3
(b) Ensemble of 3 Models
Model 1
Model 2
Model 3
(c) Diverse Ensemble
Set of Orthogonal perturbations spanning the space of the Input
Set of orthogonal adversarial perturbations for a model
Adversarial Subspace of the model/ensemble

Figure 7: Adversarial subspaces of models in an ensemble and their overlap. The challenge in an ensemble is to move from (b) to (c)

Pang et al [58] note that historically, ensemble diversity was implemented by encouraging the individual models to have different predictions for given inputs. The authors argue that this definition of diversity assumes that most individual models are weak classifiers (that they classify less than half of inputs correctly) which is often not the case nowadays, and that encouraging diversity of predictions in the classifiers will simply reduce their generalisation performance. Instead, the authors propose to train the models of the ensemble together by promoting diversity of their non-maximal predictions, arguing that this will lower the transferability between the models, likely because this diversity leads to less overlap in adversarial subspaces. The encouragement of this diversity is implemented by adding a regularisation term to the training loss function which encourages the non-maximal predictions elements of the output vector for each classifier to be as orthogonal as possible.

Kariyappa and Qureshi [57] aim to reduce the shared adversarial subspace of the models of the ensemble by using a regularizer in the training loss function called gradient alignment loss (GAL). GAL, a metric devised by the authors, measures the alignment of the gradients of the member classifiers for an input x using cosine similarity. The higher the alignment is, the higher the GAL value is and the more similar the direction which the input must move to become adversarial is for each classifier, meaning the higher the shared adversarial subspace. Therefore, by including GAL as a regularizer in the loss function used to train the network, the network is trained to minimise GAL values for inputs and the shared adversarial subspace is reduced, leading to adversarial robustness.

2.4.2. Denoising and detection

The second category of defences is denoising and detection strategies, which involve trying to remove adversarial noise. The process can be applied either to the inputs, in which case it is called input denoising, or at a hidden layer in the NN, in which case it's called feature denoising.

Xu et al [59] implemented an input denoising strategy where an image is squeezed in two different ways and then passed into the model, whilst the original image is also passed into the model. The intention of the squeezing is to remove the adversarial perturbation. The outputs corresponding to each of the 2 inputs which are squeezed are compared to the output from the original image, and if the difference is sufficiently high then the original image is deemed to be adversarial.

Another example of an input denoising strategy is MagNet, implemented by Meng and Chen [60]. MagNet consists of two components: a detector and a reformer. An autoencoder, called the reformer, is used to learn the manifold of the original images. The detector is then trained to use the manifold from the reformer and measure how far away inputs are from it; if the distance is above a threshold, the input is deemed adversarial. Such an input is then fed through the reformer and the output is expected to be a modified version of the input which is closer to the manifold learnt by the reformer, thereby removing the adversarial perturbation. GAN-based input cleansing, a third example of input denoising, works in a similar way and has had multiple implementations.

With regards to feature denoising, Liao et al [61] proposed a framework where a denoising u-net is trained to apply the transformation which minimises the difference in feature-level activations of the victim model between an inputted adversarial example and its original counterpart.

2.4.3. Gradient masking

The third category of adversarial defences is gradient masking. As most generation methods require knowledge of the gradient of the output of a network with respect to the input, gradient masking defence strategies aim to prevent the adversary from accessing this knowledge and thus prevent them from finding the adversarial subspace where adversarial examples lie.

Distillation [62], a method used to reduce the size of a DNN, was repurposed by Papernot et al [63] as an adversarial defence strategy. A new DNN with a smaller architecture is trained with the logits vectors of the original DNN acting as the labels. As part of this, when inputs are passed through the new DNN, the temperature used is the SoftMax training temperature divided by 100. By doing this, the logits vector produced for the inputs is highly sparse, meaning that the scores for the non-maximal predictions are very low. The gradients needed to increase these scores, which are needed to change the classification, become hard to form so adversarial examples can't be found.

Shattered gradient defences add non-differentiable or non-smooth layers to the victim model, which make it impossible to compute the derivative of the output with respect to the input. An example of this is the thermometer encoding applied by Buckman et al [64]. The inputs are passed through a processor which maps the pixel value into binary format, and the network is trained on the outputs of this mapping function. As the mapping function cannot be differentiated, the gradient cannot be found, and the adversary cannot move the image towards the adversarial subspace. Vanishing/exploding gradient defences work by making neural networks extremely deep, which causes the gradients to either increase or decrease exponentially in proportion to the number of layers within the network.

Athalye et al [65] deem the shattered and vanishing/exploding gradients strategies to be special cases of gradient masking called obfuscated gradients. In the same paper, they propose Backward Pass Differentiable Approximation, which involves approximating the gradient across the layer which the gradient can't be determined by using a differentiable approximation, as a technique for overcoming these defences and find it to be successful at doing so. It of it. Additionally, as pointed by Xu et al [31], an inherent weakness of gradient masking defences is that they only defend against attacks and can't prevent the existence of adversarial examples.

2.4.4. Randomisation-based schemes

The final category of adversarial defence strategies is randomisation-based schemes. The fundamental idea of randomisation-based schemes is to give the adversary different gradients to the ones that are needed to move the input towards the adversarial subspace it must reach to be sure of fooling the classifier on which it will test the final adversarial example. These methods do this by inputting randomness in the classifiers.

Two prominent examples of randomisation-based schemes are random self-ensemble (RSE) [66] and random resizing and padding [67]. In RSE, a layer adding random noise to its input channels is inserted prior to convolutional layers. To aid the network in generalisation performance when not under adversarial attack, these noise layers are included during training so that the network is somewhat adjusted to the distribution, and multiple forward passes for each input are done, so that the effect of the noise can be averaged out. In random resizing and padding, input images are resized at random and then have random amounts of padding added to them. Both works reported increases in adversarial robustness without drops in generalisation performance.

By adding randomness into the model, a situation is created where the model queried on any given query is one from several possible models, and maybe even one from an infinite number of possible models. To be guaranteed to fool the model that it will test the final adversarial example on, the adversary must find a perturbation which lies within the common adversarial subspace of all the possible models. The size of this common adversarial subspace depends on the amount of overlap between the adversarial subspaces of the possible models; the smaller the overlap between them, the harder it is for the adversary to find the common adversarial subspace. This defensive effect is reminiscent of the robustness provided by ensemble defences.

Whilst this minimisation of common adversarial subspace helps, the defining advantage of randomisation-based schemes is the gradient masking it provides. The randomness of the model means that the gradient which the adversary receives after a given query and uses to form the adversarial perturbation is the gradient towards the adversarial subspace of the model that happened to be used on that given query, as opposed to the gradient towards the common adversarial subspace, which is what the adversary needs. These two gradients are not necessarily in the same direction. Coincidentally, the aligning of these two gradients is directly dependent on the degree of overlap of adversarial subspace between the models. If there is high overlap, then these two gradients will be similar, and if there's low overlap, they will not be similar.

Bearing all the above in mind, it can be said that the determining factor in the effectiveness of randomisation-based schemes is the level of overlap between the adversarial subspaces of the possible models, much like in ensemble defences. Note, though, that unlike in ensembles classifiers, we do not induce the different possible models to have diverse adversarial subspaces; the diversity is a function of the base model to which randomisation is applied.

Just as they do with vanishing/exploding gradients and the shattered gradients defence strategies, Athalye et al [65] consider randomisation-based defence strategies as falling under gradient obfuscation, and develop a technique called Expectation-over-Transformation (EoT) to overcome the problem. As opposed to taking a single gradient for each query the adversary passes through the network, EoT means that the adversary takes multiple gradients and uses their average as the final gradient in which it moves the original image.

The main challenge facing randomisation-based schemes is that having a different model on each query could mean that the classifier will not have good generalisation when not under adversarial attack. To mitigate this effect, often multiple iterations of the forward pass are used like an ensemble.

2.5. Evaluating adversarial robustness

Measuring the pointwise adversarial robustness of an input of a machine learning model, defined in section 2.2, is a difficult task. Though it is possible to determine the ground-truth adversarial robustness of a point, it requires searching through all possible perturbations to the image for all possible perturbation sizes i.e., an exhaustive search, which is computationally demanding and often not practically possible. Having said this, a work by Carlini et al [68] tried to do it using a computational technique called Reluplex [69]. The technique is still in its infancy, though, and is not yet applicable to complex neural networks. Others [70], [71] have tried to do the same using other techniques, but their attempts make approximations [37] so their results lack reliability.

As a result, the only current option is to use a known generation technique to find an adversarial example for the input and take the perturbation of that adversarial example as the pointwise adversarial robustness. However, as none of the current generation techniques are theoretically optimal, any pointwise adversarial robustness found using this method is only an upper bound of the ground-truth pointwise adversarial robustness. The extent to which the upper bound is useful depends on the strength of the attack; the stronger the attack, the more likely it is that the true pointwise adversarial robustness lies close to the upper bound, making it a good approximation, but if the attack is weak, it could lie far below it.

On a model level, two metrics are used: worst-case adversarial robustness and global adversarial robustness, also defined in section 2.2. Worst-case robustness refers to the minimal pointwise adversarial robustness of the model over the entire input space. The issue with this metric is that since there are points lying either on or right next to decision boundaries where the smallest of perturbations will change their class, the worst-case robustness will always be zero. The adversarial robustness of these points probably doesn't matter anyway, as they are likely to be points in the input space where the combination of pixels sums to a meaningless image.

A better metric would therefore be to find the worst-case robustness over the set of all meaningful inputs. This would be the most useful metric for proving that a model can be trusted in a practical application: if the adversarial robustness is higher than whatever perturbation can be reasonably expected in the application, then the model can be used. However, calculating this would require a way to define which inputs are and aren't meaningful, which is incredibly time consuming, as well as an exhaustive search for all inputs in the final set of meaningful images, which is computationally expensive and therefore impractical in almost all situations. Therefore, the most practical metric for measuring the adversarial robustness of a model is to take a representative sample of meaningful inputs, compute the pointwise adversarial robustness for each of them and determine the average of these to find what is called the global adversarial robustness.

Another common metric used is attack success rate for a given perturbation budget. This involves restricting the adversary's capabilities to perturbations smaller than or equal to this budget and measuring the percentage of the inputs in a sample for which the adversary can find an adversarial example. The lower the attack success rate, the more adversarially robust the model. Technically speaking, as the attack success rate is given in percentage terms, it is equal to 100% minus the model accuracy on the adversarial examples, so the model accuracy can be used as an alternative to the attack success rate, with a higher model accuracy indicating greater adversarial robustness. In the present work, this metric will be referred to as adversarial model accuracy.

Measuring the adversarial model accuracy for a range of perturbation budgets produces a plot of adversarial model accuracy against perturbation budget. An example is shown in figure 8.

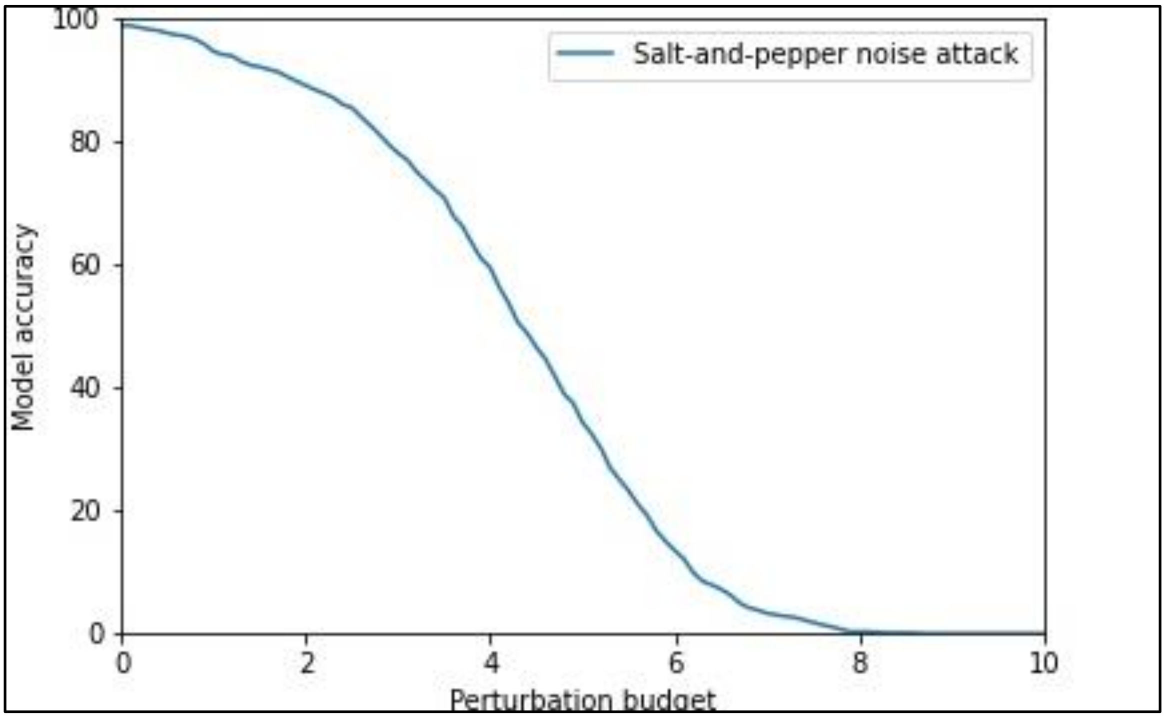


Figure 8: Plot of adversarial model accuracy against perturbation budget

Such plots are useful for having an idea of the distribution of the adversarial robustness of the individual points, and whilst this is not a single quantitative metric, it can be used qualitatively as something on which to judge the adversarial robustness of the model as a whole. Having said this, the average adversarial model accuracy over the range of perturbation budgets can be used as a metric for adversarial robustness. This is calculated by summing the adversarial model accuracy for each perturbation budget and dividing by the number of perturbation budgets evaluated. The higher the value, the greater the adversarial robustness.

Bastani et al [70] proposed two new metrics called adversarial frequency and adversarial severity. Adversarial frequency is the probability that a point in the input space is ε-robust, mathematically written as:

$$\Phi(F, \varepsilon) = P(\rho(F, x) \leq \varepsilon) \tag{22}$$

Adversarial severity measures the expected robustness of points which aren't ε-robust to give an indication of how severely the classifier falls short of being ε-robust, given mathematically as:

$$\mu(F, \varepsilon) = E(\rho(F, x) | \rho(F, x) \leq \varepsilon) \tag{23}$$

However, neither metric seems to have been used commonly in the literature and neither are used in the present work.

2.6. Test-time dropout

Test-time dropout refers to randomly dropping out nodes or channels from the neural network in its inference phase, and its most common usage has been in estimating the uncertainty of model predictions.

2.6.1. MC dropout

Bayesian deep neural networks (BDNNs) are DNNs where each weight is associated with a probability distribution instead of a value. An example is shown in figure 9 [72]. The added complexity of having a distribution instead of a value means that multiple methods exist for conducting inference of an input using a BDNN [73], such as variational inference and sampling.

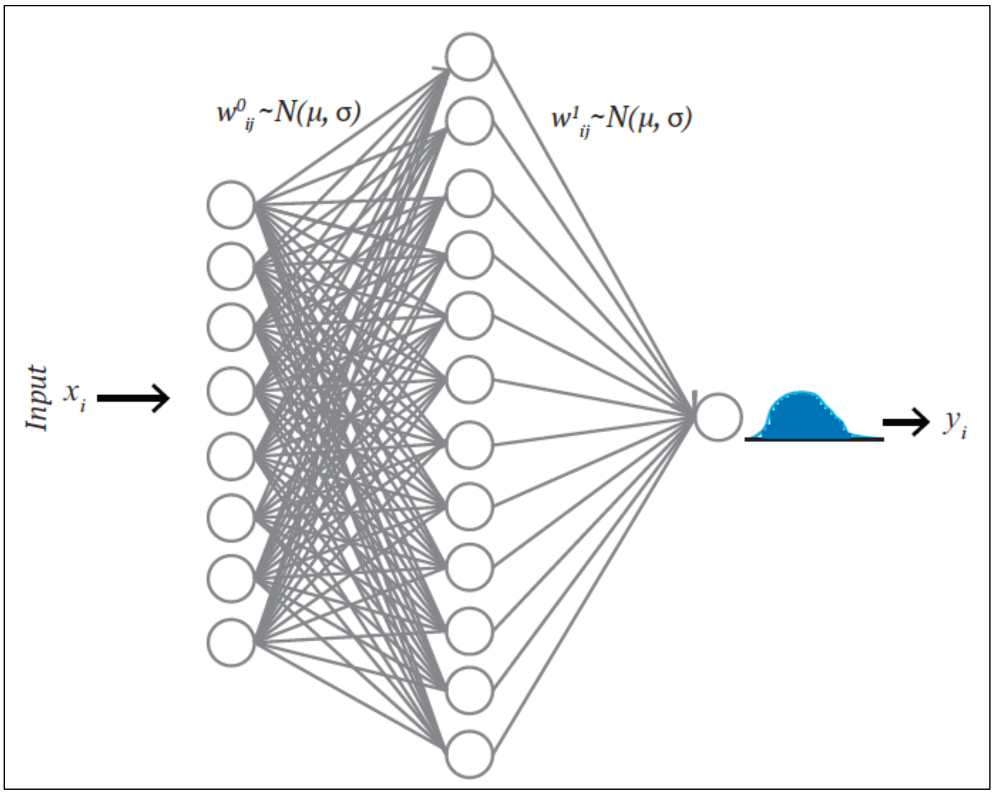

$w^0_{ij} \sim N(\mu, \sigma)$
$w^1_{ij} \sim N(\mu, \sigma)$
Input
x_i
y_i

Figure 9: Architecture of a typical BDNN

To derive a classification from a BDNN for an input using sampling, multiple forward passes of the input are done. At each forward pass, the value taken for each weight is a random variable selected from its associated distribution. This results in the output for each node being a distribution of values rather than a single value. The final value of an output node for an input is taken as the mean of the distribution. The variance of the distribution of the network can also be calculated, and this is where the main advantage of BNNs over conventional DNNs is: the value of the variance can be used as an estimation of the uncertainty of the model's prediction. High variance means that the network is less certain of its value for the output, whilst low variance means it is more certain. A drawback of BNNs, though, is that they are computationally expensive and time consuming to train and are therefore often impractical.

As a solution to these problems, in 2016 Gal and Ghahramani [74] developed a theory explaining how using dropout at test time on a conventional DNN trained using dropout can be interpreted as an approximation of a Gaussian process and thus is almost the same as the sampling method used for BDNN inference. By using multiple forward passes for a test input with a random dropout configuration applied each time, a distribution of values for each output node is obtained and can be used in the same way in which it's used for a BNN, thus allowing model uncertainty to be determined. The theory has become widely accepted in the field, and the name given in the literature to the technique is Monte Carlo (MC) dropout [75], as it can be thought of as using Monte Carlo simulations for the estimation of the mean and variance statistics of the distribution of each output node.

MC dropout has seen success in several applications. Cortes-Ciriano and Bender [76] combined MC dropout with conformal predicting in the field of drug discovery. The authors found that their strategy efficiently generated valid conformal predictors and confidence intervals comparable to those generated using Random Forest-based conformal predictors. Durr et al [77] used MC dropout to define multiple uncertainty measures in the context of a phenotype classification task. They found that the technique was useful in identifying novel phenotypes not present in the training data as the output nodes for these phenotypes displayed high variability. Coincidentally, they also found that MC dropout improved the overall accuracy of the model.

Brach et al [78] noted that a drawback of MC dropout is that it requires multiple forward passes, which increases runtimes and means that it may not be practical in certain situations. To solve this, they propose to use statistical moment propagation to approximate the variance of the signal produced by each node had full MC dropout been used, calling it 'Single-shot' approximation.

In 2017, Feinman et al [79] attempted to use MC dropout in the context of adversarial examples. They started with the assumption that adversarial examples lie off the data submanifold of their true class. Based on this, they argued, and later proved, that the variance of predictions for adversarial examples should be higher, and that therefore the variance of an input could be used to detect whether the input was adversarial. Such a strategy falls under the denoising and detecting category.

2.6.2. Test-time dropout as a randomisation-based scheme defence strategy

More recently, the use of test-time dropout has been reviewed in the literature as a randomisation-based scheme defence strategy. Note that in such works it is not referred to as MC dropout for two reasons. Firstly, in these works, the technique of test-time dropout is not used to determine the uncertainty of a prediction, and secondly, single forward passes are used for prediction instead of multiple forward passes. These details aren't included because in a randomisation-based scheme, the goal is to confound an adversary by depriving it of useful gradients, so neither detail is required.

The defensive effects of test-time dropout used as a randomisation-based scheme can be explained as follows. By dropping out n nodes or channels and their connections from the model at random, 2^n possible models can be formed, and at each query, one is selected by virtue of the Bernoulli trial which each node or channel undergoes. As the model at each query is different, the gradients taken at each query do not necessarily align with the gradients needed to move the input to the common adversarial subspace of all 2^n possible models, thereby hindering the adversary from finding the adversarial examples needed to ensure fooling the final query classifier.

Dhillon et al [80] proposed using a technique similar to dropout during inference which they called stochastic activation pruning (SAP). In SAP, at each layer l, in each activation map, a subset of activations is drawn from the set of all the activations in that map using random sampling with replacement. Each activation's probability of being drawn is proportional to the magnitude of its activation. Therefore, on a given sampling trial, the probability of activation $z_{j,i}^{l}$ being drawn is

$$p_{j,i}^{l} = \frac{\left|z_{j,i}^{l}\right|}{\sum_{i=1}^{n}\left|z_{j,i}^{l}\right|} \tag{24}$$

where j refers to the activation map in that layer, i refers to the activation within the activation map, and n refers to the number of activations in the activation map. If an activation is not sampled, it's set to 0. If it is, it's scaled up in proportion to its magnitude. This differs from traditional dropout, where the inclusion of each activation is decided by an independent Bernoulli trial with a predefined, uniformly applied success probability p. Despite SAP being one of the techniques criticised by Athalye et al [65] as being an obfuscated gradient defence which they claimed to have broken using BDPA, a later work by Dhillon and Carlini [81] found that their tests had not been applied faithfully.

Wang et al [82] used conventional dropout during inference using two CNNs, one trained for the MNIST [83] task and the other for CIFAR-10 [84]. The authors tested various combinations of training-time and test-time dropout rates using a defensive hardening algorithm which aimed to find the combination which optimised performance both when under adversarial attack and when not. The general trend found was that as test-time dropout rate increased, test accuracy decreased but attack success rate increased, and as training-time dropout rate increased, test-time accuracy increased but attack success rate increased.

Wang et al [82] also demonstrated a result proving that the rationale given above as to why test-time dropout works as a randomisation-based scheme, which resembles their own explanation, is at least partially correct. They found that the distribution of the gradients that an adversary using an iterative generation technique received corresponding to a given pixel in the image over the execution of an attack got shorter and fatter as the dropout rate increased. This can be explained as follows. As the model at each query is slightly different and thus has a different adversarial subspace, the set of gradients received will have more variability than in the case of a single deterministic model, as they will be pointing in slightly different directions each time. As well as this, as the dropout rate increases, the probability of a given model from the set of all possible models being chosen changes, meaning that the distribution describing the probability of each model being chosen also becomes shorter and fatter. As a result, the variability of the model chosen increases, meaning that the variability of the gradients increases even more.

2.7. Knowledge gaps

Despite all the work on the application of test-time dropout to adversarial attacks, there remain knowledge gaps in the subject area. The following list enumerates some of them:

1. The victim model architecture used by Wang et al [82], which will be discussed in section () of the present work, is quite basic, so the results may not be representative of how test-time dropout will work in the more complicated architectures used in most applications.
2. As mentioned in the discussion of defence strategies, the diversity of the possible models in a randomisation-based scheme is dependent only on the underlying model. None of the work done so far on test-time dropout has addressed how to train the underlying model to encourage the diversity of the possible models.
3. All existing works use existing generation techniques when evaluating adversarial robustness, as opposed to using methods to measure the objective adversarial robustness.
4. Wang et al [82] only applied test-time dropout on the fully connected layers, and never to the channels produced by convolutional and pooling layers.

5. Dhillon et al [80] applied test-time dropout to specific activations within a channel/activation map, and not to an entire activation map.
6. Ensembling has not been used in any previous test-time dropout work to improve generalisation performance, so it is unknown whether it can be used to help generalisation without compromising adversarial robustness.
7. The reporting of results by Wang et al [82] is scattered and not comprehensive, and no resources are provided where full results are stored or where files are stored for recreating the experiment.

Knowledge gaps 1-3 will not be addressed by the present work due to them being outside of the scope of the project as well as due to time constraints. Instead, it will aim to build on the work of Wang et al [82] and Dhillon et al [80] by filling in knowledge gaps 4-7. Details as to how these gaps are filled are given in section 3.

3. Experimental setup

Carlini et al [35] listed several criteria which they believe experimental setups for evaluating adversarial robustness should strive to meet to make their results reliable, reproducible, and amenable to useful and honest comparison with other works. The list is extensive and yet by the authors' own admission [35] not exhaustive. The present work has tried to follow its guidelines in devising and describing the experimental setup but acknowledges that it does not do so perfectly. The experimental setup also has other shortcomings, some of which are discussed in section 5.3.

An experimental setup for evaluating adversarial robustness is composed of four parts: the victim model, the adversary's threat model, the defensive strategies used by the victim model, and the metrics used to evaluate adversarial robustness.

3.1. Software

Before explaining the details of the setup, the computational tools for implementing it will be mentioned. The programming language used will be Python [85]. PyTorch [86] is used for implementing the datasets, DNNs and defence strategies, and FoolBox [87] is used for implementing the various adversarial attacks. As generating the adversarial examples is computationally demanding, access to graphical processing units (GPUs), as opposed to computational processing units (CPU) on conventional PCs, is required, so Google Colaboratory Pro [88] will be used as the source code development environment. Examples of the code created for implementing the experimental setup using these tools can be seen in appendices 10-12.

3.2. Victim model

The victim model is defined by its application domain, the dataset its trained on and task its trained to complete, its architecture and its training algorithm. The defence strategies literature review [23], [37], [50], [52], [58], [63], [66], [67], [82] showed that the most common domain for testing adversarial robustness is image classification, which is likely because this is the domain where the phenomenon was discovered, and which is the most applicable industrially. Only the ImageNet, MNIST, CIFAR-10 and CIFAR-100 datasets were used amongst those works, and most used some form of residual neural network (ResNet) [89], such as Inception-v3, for the model architecture.

The victim model design choices for the present work were made with comparison with these works in mind, particularly with the work of Wang et al [82] as it's the most recent test-time dropout work. The domain of image classification is chosen, specifically the MNIST [83] dataset. Although ResNets were used in most works, this was for the CIFAR-10 and CIFAR-100 datasets; for the MNIST task, simpler architectures were used. Papernot et al [63], Carlini and Wagner [37] and Wang et al [82] all used the same architecture for the MNIST task, with each work referencing comparison with the others as reason for choosing this architecture. To facilitate comparison with all these works, this same architecture, detailed in table 1, is used in the present work, and the underlying model is trained in the same way, as detailed in table 2. The loss function used was cross-entropy loss, and the optimiser used was stochastic gradient descent with a momentum term.

Whilst the model was trained on the entire MNIST training dataset, i.e., all 60000 images, only the first 1000 images of the 10000-image test dataset were used for generating adversarial examples. No normalisation was applied to the images.

Layer type	Channels/activations in	Channels/activations out	Details
Convolutional + ReLU	1	32	3x3 kernel for each channel
Convolutional + ReLU	32	32	3x3 kernel for each channel
Max Pooling	32	32	2x2 kernel with stride of 2
Convolutional + ReLU	32	64	3x3 kernel for each channel
Convolutional + ReLU	64	64	3x3 kernel for each channel
Max Pooling	64	64	2x2 kernel with stride of 2
Fully connected + ReLU	1024	200	Dropout applied to this layer during training
Fully connected + ReLU	200	200	N/A
Fully connected + SoftMax	200	10	N/A

Table 1: Victim model architecture

Parameter	Value
Learning rate	0.1
Momentum	0.9
Dropout rate	0.5
Batch size	128
Epochs	50

Table 2: Victim model training details

3.3. Threat model

As mentioned earlier, the threat model is defined by the adversary's goal, knowledge, and capabilities.

3.3.1. Goal

Due to the complexity of implementing targeted attacks, only untargeted attacks will be used.

3.3.2. Knowledge

With regards to the adversary's knowledge of the victim model, the present work assumes that the adversary has complete knowledge of the topology and parameters of the underlying model and can access gradient information from it i.e., a white-box scenario. Usually, if the defence strategy is proven to be effective in this scenario, it can be assumed to be effective in black-box scenarios. However, for defence strategies using some form of gradient masking, this cannot be assumed, as the mechanism of gradient masking works in such a way that any white-box scenario is essentially reduced to a black-box one. Based on this, Carlini et al [35] included a criterion that states that for models with defence strategies which use some form of gradient masking, both white-box and gradient-free attacks should be tested. As the explanation provided in section 2.6.2 for why test-time dropout might be effective is based on gradient masking, both types of generation methods should be included in the present work. If the results show that black-box attacks are not outperformed by white-box attacks, it will provide evidence for the argument that any defensive effects observed from test-time dropout are due to gradient masking.

The specific generation methods used and the hyperparameter values for each of them are listed in table 3. For the white-box attacks, all generation methods are implemented using both the L_2-norm and L_∞-norm, besides for the Carlini-Wagner [37] attack, which only uses the L_2-norm because an L_∞-norm implementation is not offered by FoolBox [87]. As the Carlini-Wagner [37] attack incorporates a constant c, as explained in section 2.3.1, a binary search must be done to find its best value. For this binary search, 20 steps are done with an initial c value of 0.01.

Generation method	Step size (relative to perturbation budget)	Number of steps
Fast-gradient sign method ($\boldsymbol{L_2}$)	N/A	N/A
Projected gradient descent ($\boldsymbol{L_2}$)	1/250	260
DeepFool ($\boldsymbol{L_2}$)	N/A	260
Carlini-Wagner ($\boldsymbol{L_2}$)	0.01	10000
Fast-gradient sign method ($\boldsymbol{L_\infty}$)	N/A	N/A
Projected gradient descent ($\boldsymbol{L_\infty}$)	1/250	260
DeepFool ($\boldsymbol{L_\infty}$)	N/A	260
Salt-and-pepper noise ($\boldsymbol{L_2}$)	N/A	N/A

Table 3: Generation methods used and their hyperparameters

With regards to the black-box attacks, no implementations of any score-based attacks are available in FoolBox [87], and for decision-based attacks, the Boundary attack [46] was not used due to difficulties implementing it computationally. Every time it was run, it would return an error stating that an adversarial starting point could be not found for some of the input images, and no option is provided to run the attack with only those inputs which a starting point could be found for. Instead, the salt-and-pepper noise attack is implemented.

In the present work, the adversary is unaware of the application of dropout and is also unaware that ensembling is applied for classifying the final adversarial example. This prevents the adversary from implementing counter-attack strategies.

3.3.3. Capabilities

For all generation methods, a range of perturbation budgets are applied, as opposed to a single limit, and the adversarial robustness evaluation is applied for each of them. This is helpful for comparison with other works, where only one perturbation budget may be applied. Also, doing this facilitates the plotting of adversarial attack success rate against perturbation budget, which as explained earlier helps to give a broader picture of the adversarial robustness of a DNN.

For attacks using the L_2-norm, a perturbation budget range of 0-10 is used, applied in increments of 0.1. For those using the L_∞-norm, a perturbation budget range of 0-1 is used, applied in increments of 0.01. These ranges were decided upon through trial-and-error. For ranges with a smaller maximum value, the adversarial model accuracy did not reach close to 0% for any generation method for the underlying model both with and without test-time dropout applied, which would make any adversarial robustness evaluation done with such results incomplete. For ranges with a higher maximum, the runtimes were too high and, for the undefended underlying model and most of the defensive configurations, at least one attack had reached 0% adversarial model accuracy before reaching the maximal perturbation budget meaning that applying the higher perturbation budgets didn't add any information.

The meaning of a perturbation budget for the FGSM attack is simply the step size ε applied, as defined in section 2.3.1. For the salt-and-pepper noise attack, the perturbation budget refers to the maximum amount of noise that can be added prior to classifying the example. For all the other attacks, which are iterative, the perturbation budget is applied by initially allowing the adversary to apply a perturbation whose magnitude is equal to the maximum perturbation allowed (which is 1 or 10 in this case) and then clipping this perturbation to be within the perturbation budget. This is equivalent to projecting the maximally perturbed image onto the surface of the ε-neighbourhood of the initial image x to produce x'.

3.4. Defensive strategy and configurations

3.4.1. Strategies used

Whilst some works implement multiple defensive strategies simultaneously, the present work applies test-time dropout on its own. As work concerning the effectiveness of test-time dropout is still in the early stages, it's best to evaluate on its own so that any defensive effects are known to come from its application and not from any other strategies implemented.

3.4.2. Test-time dropout configurations

Knowledge gaps 4 and 5 state that test-time dropout has never been applied to a convolutional layer at an activation map level, so the present work will do this. Test-time dropout will be applied to each layer individually, besides for the output layer. For each layer, test-time dropout rates of 0.1, 0.3, 0.5, 0.7 and 0.9 will be tested. As 5 dropout rates will be applied to 8 different layers, 40 test-time dropout configurations are tested in total.

3.4.3. Ensembling

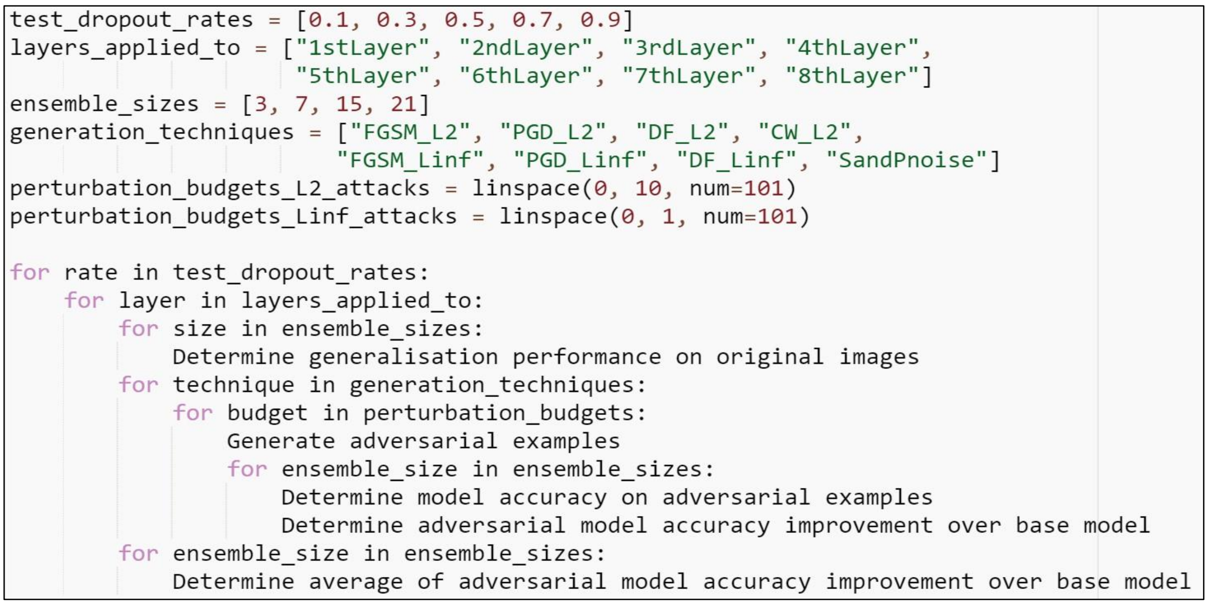
To mitigate the generalisation issues which randomisation-based schemes often face, test-time dropout will be applied multiple times, as an ensemble, with majority voting used as the combination function for determining the final prediction. For each test-time dropout configuration, ensemble sizes of 3, 7, 15 and 21 are used. Doing this will address knowledge gap 6.

Section 2.4.1 discussed how ensembling can be used as a defence strategy on its own. When the ensemble is made up of the same set of classifiers on each forward pass, it always reduces the adversarial subspace of the final classifier. However, if a randomisation-based scheme is already in place, the set of classifiers used in the ensemble is randomly chosen and thus not always the same. As a result, adding ensembling to a randomisation-based scheme can feasibly have a negative effect on adversarial robustness just as much as it can have a positive effect.

For example, when using an ensemble of 15 randomly chosen models for the final query, the example would need to fall within the common adversarial subspace of at least 8 of them in order to fool the classifier. Knowing whether the probability of this occurring is higher or lower than the probability of the input falling in the adversarial subspace of a single model that would be randomly chosen from all the possible models requires knowing all possible 15-model combinations and knowing which of them have overlapping adversarial subspaces around the input, which is practically unobtainable knowledge. Therefore, when applying ensembling within the experiments of the present work, there is no intention of using it to improve adversarial robustness, and the only intention is to use it to improve generalisation. Whilst the effect of ensemble size on adversarial robustness will be analysed, the present work proposes no rationale as to why these effects may have occurred.

In the present work, ensembling is only applied for the final query, when the final adversarial example is passed to the DNN; for queries in the adversarial example generation process, only one forward pass is used. In a real-life application, ensembling would need to be applied on every query of the DNN, as the DNN would not know whether the query was being used as part of the generation process or as the final query. Therefore, this setup is not reflective of a real-life application, and is only implemented in this way in the present work due to FoolBox not allowing multiple queries to be done during the generation process.

3.5. Metrics used



```
test_dropout_rates = [0.1, 0.3, 0.5, 0.7, 0.9]
layers_applied_to = ["1stLayer", "2ndLayer", "3rdLayer", "4thLayer",
                     "5thLayer", "6thLayer", "7thLayer", "8thLayer"]
ensemble_sizes = [3, 7, 15, 21]
generation_techniques = ["FGSM_L2", "PGD_L2", "DF_L2", "CW_L2",
                         "FGSM_Linf", "PGD_Linf", "DF_Linf", "SandPnoise"]
perturbation_budgets_L2_attacks = linspace(0, 10, num=101)
perturbation_budgets_Linf_attacks = linspace(0, 1, num=101)

for rate in test_dropout_rates:
    for layer in layers_applied_to:
        for size in ensemble_sizes:
            Determine generalisation performance on original images
        for technique in generation_techniques:
            for budget in perturbation_budgets:
                Generate adversarial examples
                for ensemble_size in ensemble_sizes:
                    Determine model accuracy on adversarial examples
                    Determine adversarial model accuracy improvement over base model
        for ensemble_size in ensemble_sizes:
            Determine average of adversarial model accuracy improvement over base model
```

Figure 10: Pseudocode of how the results will be produced from the experimental setup

Section 2.5 mentioned several metrics that can be used to measure adversarial robustness. In the present work, the metric chosen is the average adversarial model accuracy.

Two steps are used to calculate this final metric. First, the adversarial model accuracy will be determined for each perturbation budget in the perturbation budget range for each combination of generation method, test-time dropout configuration, and ensemble size. This will enable the formation of a plot of adversarial model accuracy against perturbation budget for each combination of these 3 variables. Then, the average adversarial model accuracy over the full range of perturbation budgets is calculated for each combination of these 3 variables, to produce the final value for the average adversarial model accuracy metric.

To determine whether test-time dropout is effective at improving the average adversarial model accuracy, the values produced for the metric when test-time dropout is in place must be compared with the values produced for the metric when test-time dropout is not in place. Therefore, the same two steps mentioned above will also be done for the underlying victim model, and for each test-time dropout configuration, the difference between its average adversarial model accuracy value and that of the underlying victim model without the defence in place will be calculated. This is the final metric that will be reported for evaluating the effect of test-time dropout on adversarial robustness.

For the white-box generation methods, besides for analysing the improvement in average adversarial model accuracy for each method on its own, the results for each of them can be compared against those of the salt-and pepper noise attack. As mentioned earlier in this section, if the white-box methods do not outperform the black-box methods against the test-time dropout defence strategy, then it is possible that gradient masking is occurring, which would make test-time dropout effective at improving adversarial model accuracy and thus adversarial robustness.

As randomisation-based schemes are known to impact generalisation performance, the model accuracy on the original inputs will be calculated as well, for each combination of ensemble size and test-time dropout configuration. This will then be compared to the generalisation performance of the underlying model without test-time dropout in place, to determine the relative impact of test-time dropout on generalisation performance. Figure 10 shows the pseudocode for how the results will be produced from the experimental setup.

3.6. Outputs produced

In addition to the source code used to implement the experimental setup, the implementation of the experimental setup will produce many outputs, including the adversarial examples generated for each test-time dropout configuration, the numerical results for average adversarial model accuracy, numerical results for generalisation performance and plots of adversarial model accuracy against perturbation budget. To include all of these in the appendices of the present work would have been excessive, so instead all the outputs produced are stored at the following URL: https://drive.google.com/drive/folders/1hSr5gB4f13-TfCke6tR0Oyy-xWlanpBF?usp=sharing, with most of the folders and files also being included in the supplemental file submitted along with the present work. The only folder present on the URL but not included in the supplemental file is the one titled ‘adversarial examples’, as it was too large to be included. A full explanation of all the folders and files is found in the README.txt file in the supplemental file. Readers of the present work are encouraged to look at the supplemental file and visit the URL to explore the outputs produced.

4. Results

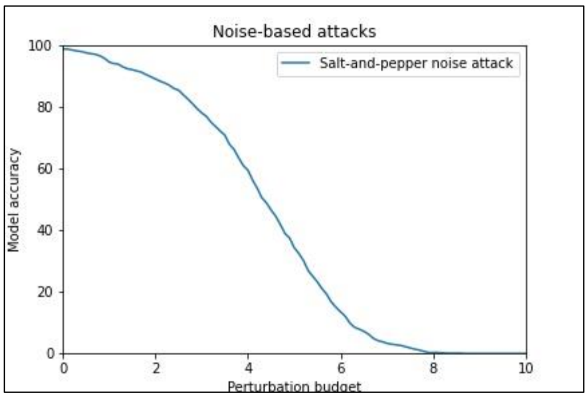
As mentioned in section 3, there are two components of the results produced by the experimental setup: the improvement in adversarial robustness provided by each test-time dropout configuration, which is measured using improvement in average adversarial model accuracy, and the impact on generalisation performance of applying test-time dropout when not under adversarial attack, which is measured using conventional model accuracy. Both will be discussed in this section.

4.1. Adversarial robustness

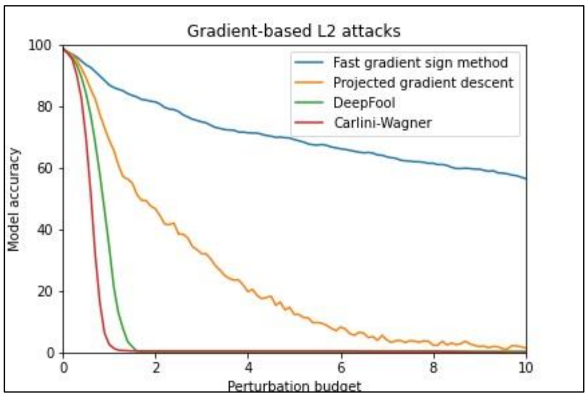
Before starting this discussion, note that in this section the term 'adversarial robustness' is not used to refer to the specific metric with the same name used for evaluating the performance of an ML model or defence strategy against adversarial attacks. Rather, it refers to the general concept of resistance against adversarial attacks, which can be measured using various metrics, as discussed in section ().

The FGSM, PGD and DeepFool generation methods were implemented using both the L_2- and L_∞-norms. However, the trends discovered for the results when defending against each of these generation methods were the same for both implementations, so only the L_2-norm implementations are discussed. Tables containing the improvement in adversarial model accuracy against all generation methods are found in appendices 1-8, whilst the raw results are included in the supplemental file submitted. This comprehensive reporting of results addresses knowledge gap 7.

4.1.1. Undefended underlying victim model



The average adversarial model accuracy for the underlying undefended victim model is provided in table 4, and plots of the adversarial model accuracy against the perturbation budget for each generation method are shown in figure 11. We observe that most gradient-based attacks are more successful than the black-box salt-and-pepper noise attack against the underlying undefended victim model. Note that ensembling is not used for the underlying undefended victim model as the model is deterministic.



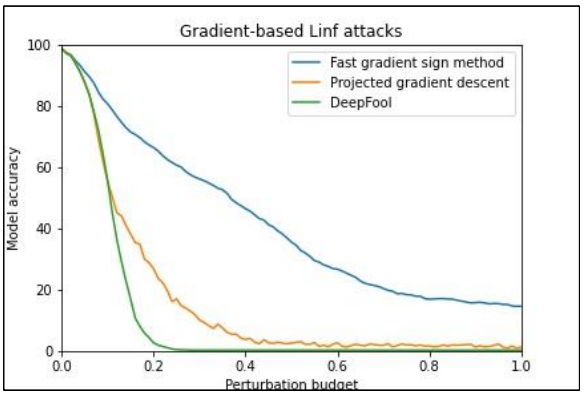


Figure 11: Plot of adversarial model accuracy against perturbation budget for the underlying undefended victim model against various attacks

Average adversarial model accuracy	FGSM L_2	PGD L_2	DeepFool L_2	Carlini-Wagner L_2	FGSM L_∞	PGD L_∞	DeepFool L_∞	Salt-and-pepper noise
	71.22	25.17	9.25	6.83	42.3	16.32	11.27	42.31

Table 4: Average adversarial model accuracy of the underlying undefended victim model against various attacks

4.1.2. FGSM $\mathbf{L_2}$ generation method

Appendix 1 contains the improvement in average adversarial model accuracy against the FGSM L_2 generation method when test-time dropout is implemented. For the 1st and 4th layers, adversarial robustness worsens as the dropout rate increases, whereas for the 7th layer, it improves as the dropout rate increases. For all other layers, adversarial robustness improves as the dropout rate increases, except for the dropout rate of 0.9, where adversarial robustness significantly worsens. The 2nd and 3rd layers have the most volatile performance, as for some test-time dropout configurations, they have the highest improvement in adversarial robustness, yet for dropout rate of 0.9, they also have the highest decrease. These trends are visualised in figure 12. For all combinations of test-time dropout rate and layer which it is applied to, increasing the ensemble size slightly improves the adversarial robustness.

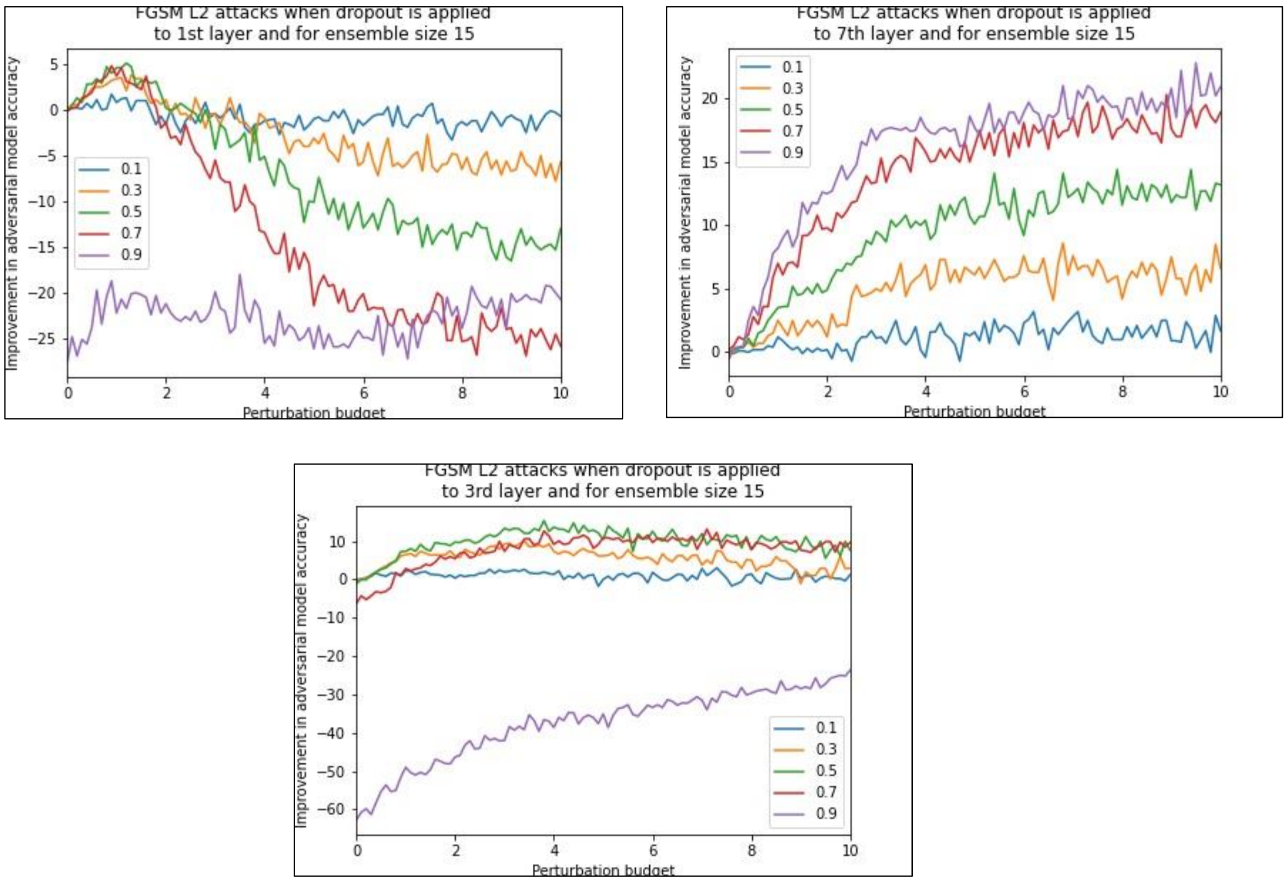


Figure 12: Plots of improvement in adversarial model accuracy against perturbation budget against the FGSM L2 attack for varying dropout configurations

Overall, these results and the trends within them indicate that test-time dropout does not provide improved adversarial robustness against the FGSM L_2 attack. There is no single test-time dropout configuration that has an improvement of 20% or greater, whilst there are many configurations where adversarial robustness is worsened by test-time dropout. As well as this, if gradient masking was occurring, as was the intention of applying test-time dropout, the effect of varying the dropout rate would be similar for each layer, which isn't the case.

4.1.3. PGD L_2 generation method

The results for the effectiveness of test-time dropout against the PGD L_2 attack, contained by appendix 2, don't show any strong, consistent trends with respect to changes in the test-time dropout configuration variables. In the 7th layer, as was the case against the FGSM L_2 attack, the improvement in average adversarial model accuracy increases as the dropout rate increases. In the 2nd and 3rd layers, all test-time dropout configurations worsen the adversarial model accuracy besides for those with the dropout rate of 0.7. Other than for these specific configurations, though, the variation of the dropout rate doesn't have a consistent effect on the average adversarial model accuracy. Just like the FGSM L_2 attack, for all combinations of test-time dropout rate and layer which it is applied to, increasing the ensemble size slightly improves the adversarial robustness. However, figure 13 shows that this mainly occurs over the lower perturbation budgets, and the trend reverses for higher budgets. One noticeable departure, though, from the results of the FGSM L_2 attack is that magnitudes of the changes in adversarial model accuracy are smaller.

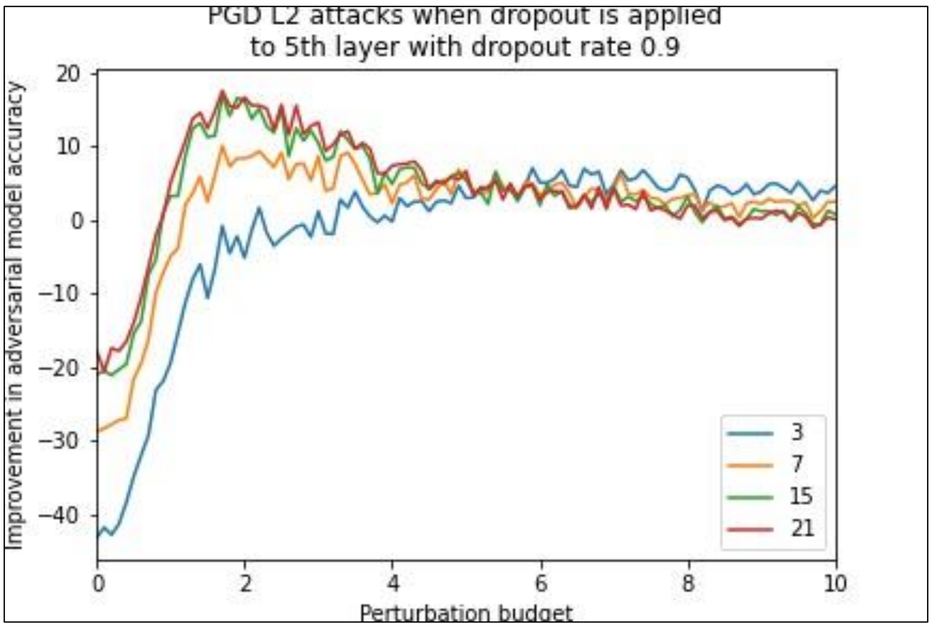


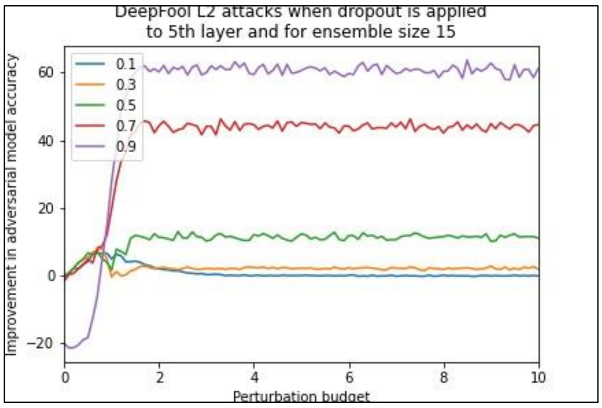
Figure 13: Plot of improvement in adversarial model accuracy against perturbation budget against the PGD L2 attack for varying ensemble sizes

The lack of strong trends combined with the lower magnitudes of improvement in average adversarial model accuracy again indicate that that test-time dropout does not provide improved adversarial robustness. The same argument regarding gradient masking that was made for the FGSM L_2 attack results applies to these results.

4.1.4. DeepFool L_2 attack

The DeepFool L_2 generation method is the first attack to provide evidence that test-time dropout is effective at improving adversarial robustness. For every test-time dropout configuration, the improvement in average adversarial model accuracy over the undefended underlying model is positive, and for some configurations, the magnitude of this improvement is significant.

Stronger trends are also observed with respect to the parameters of test-time dropout configuration. For all the layers besides for the 2nd and 3rd layers, an increase in the dropout rate leads to greater improvement in adversarial robustness. For the 2nd and 3rd layers, this trend is true up to the dropout rate 0.7, as at a dropout rate 0.9, the improvement in adversarial robustness decreases sharply. These trends can be seen in figure 14, and they act as evidence for gradient masking. As discussed in section 2.6, increasing the dropout rate leads to an increase in the variability of the model chosen at each query, which in turn leads to more variability in the gradients received by the adversary. This lack of consistent gradients is a form of gradient masking, which provides improved adversarial robustness.



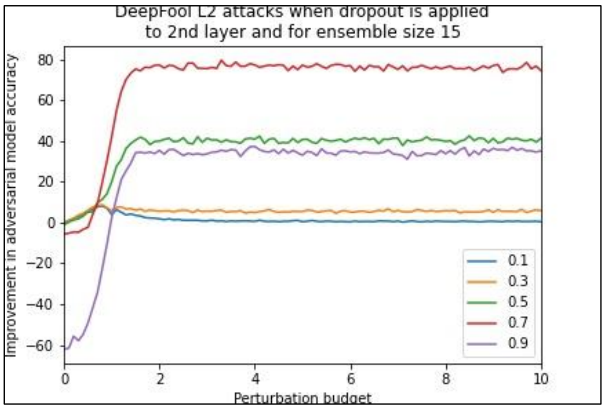
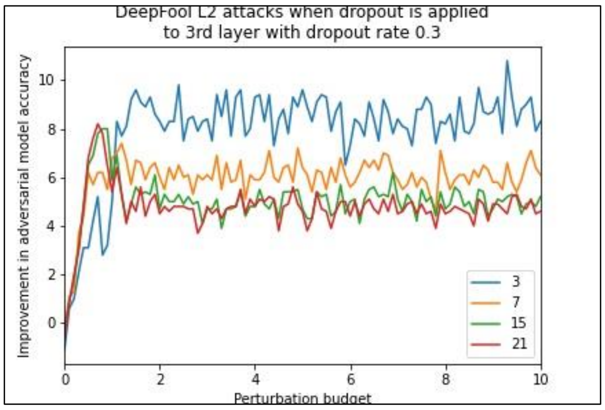


Figure 14: Plot of improvement in adversarial model accuracy against perturbation budget against the DeepFool L2 attack for varying dropout configurations

The impact of the ensemble size on the improvement in adversarial robustness depends on the dropout rate. For lower dropout rates, increasing the ensemble size slightly reduces adversarial robustness, whereas for higher dropout rates, it leads to increasing adversarial robustness. These trends can be seen in figure 15.



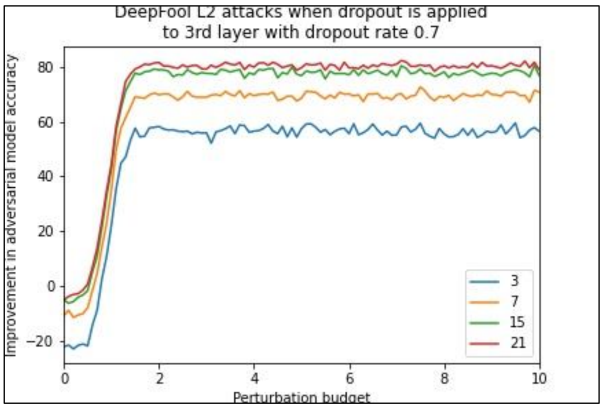


Figure 15: Plot of improvement in adversarial model accuracy against perturbation budget against the DeepFool L2 attack for varying ensemble size and dropout rate

4.1.5. Carlini-Wagner L_2 attack

The experimental results in appendix 4 indicate that test-time dropout is highly effective for improving robustness against the Carlini-Wagner L_2 generation method. Figure 16 show that the adversarial model accuracy is vastly improved over the undefended underlying model over the full range of perturbation budgets and for almost all test-time dropout configurations.

For dropout rates between 0.1 and 0.7, the effectiveness of test-time dropout does not vary with changes in the dropout rate. However, for almost all test-time dropout configurations, the defence becomes less effective at a dropout rate of 0.9. This drop in performance is most pronounced in the 2nd and 3rd layers, and least pronounced in the 7th and 8th layers. With regards to the impact of ensemble size, the results indicate again that at dropout rates between 0.1 and 0.7, the ensemble size is irrelevant, but at a dropout rate of 0.9, higher ensemble sizes provide increased adversarial robustness. Generally speaking, the effectiveness of test-time dropout against the Carlini-Wagner L_2 attack shows little variation with changes in test-time dropout configurations.

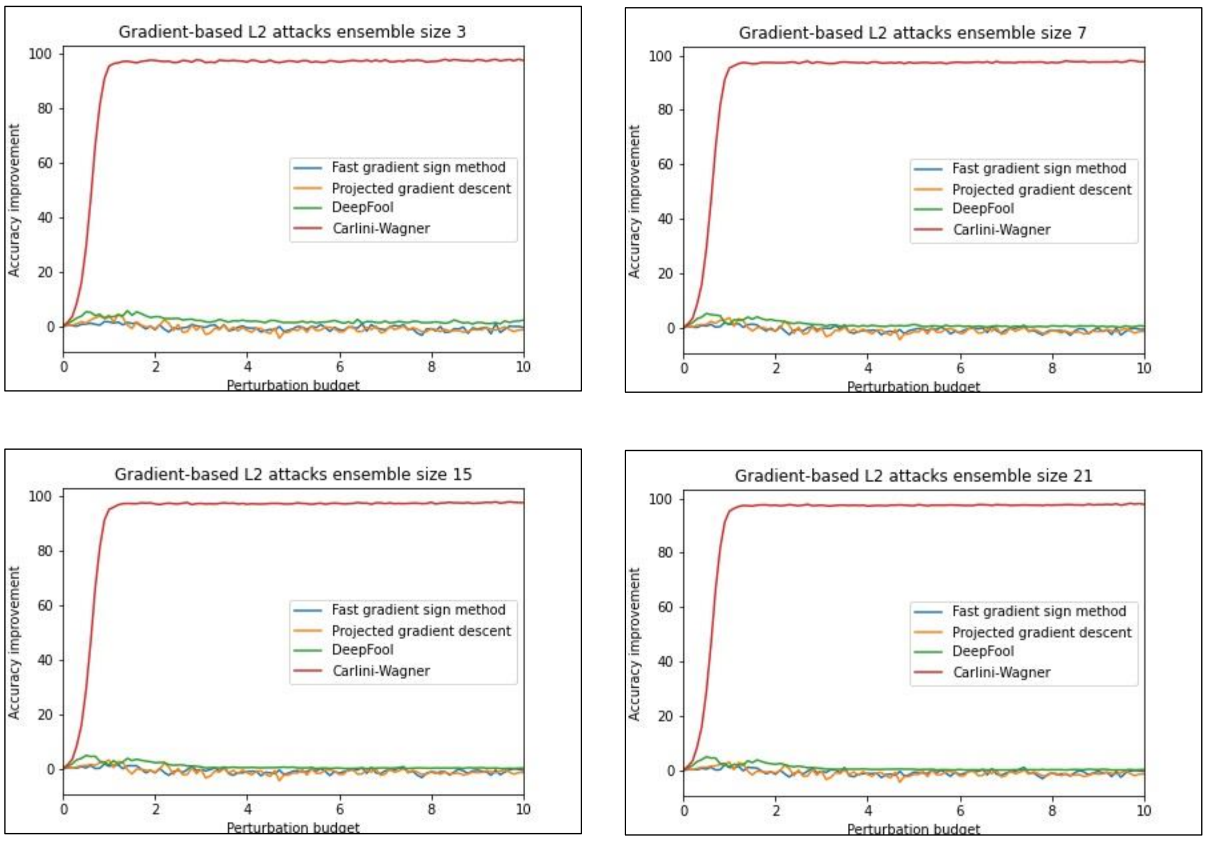


Figure 16: Plot of improvement in adversarial model accuracy against perturbation budget against the Carlini-Wagner L2 attack for varying ensemble sizes

Despite the good performance, the lack of variation provides evidence against the presence of gradient masking. If gradient masking were occurring, the effectiveness of test-time dropout against the Carlini-Wagner L_2 attack would certainly vary with the dropout rate, as happens against the DeepFool L_2 attack. As well as this, figure 16 shows that test-time dropout is vastly more effective against the Carlini-Wagner L_2 attack than against any other gradient based-attack, and if gradient masking was occurring, this would not be so. Therefore, it is more probable that the implementation of test-time dropout has somehow computationally broken the FoolBox implementation of the Carlini-Wagner L_2 attack, as opposed to fundamentally stifling the underlying attack.

4.1.6. Salt-and-pepper noise attack

The results for the salt-and-pepper noise attack are in appendix 8. The purpose of applying the salt-and-pepper noise attack was to compare how its performance against test-time dropout with that of the white-box attacks. Figures 17 shows this comparison. Note that comparison was not done with Carlini-Wagner L_2 attack as it is believed to be ineffective for unknown reasons and so it is best not to draw comparisons to it. From this comparison, we observe that, for most test-time dropout configurations, the white-box attacks remain more effective at deceiving the classifier than the salt-and-pepper noise attack. This would indicate that gradient masking isn't occurring.

However, it should be noted that for almost all test-time dropout configurations tested, there is significant improvement in the adversarial robustness against the salt-and-pepper noise attack. This defensive effect cannot possibly come from gradient masking, as the salt-and-pepper noise attack is a black-box attack. Based on this, one could argue that whatever mechanism is causing this improvement may also be causing the improvement seen against the DeepFool L_2 attack, as opposed to gradient masking, which other results provide evidence of not occurring.

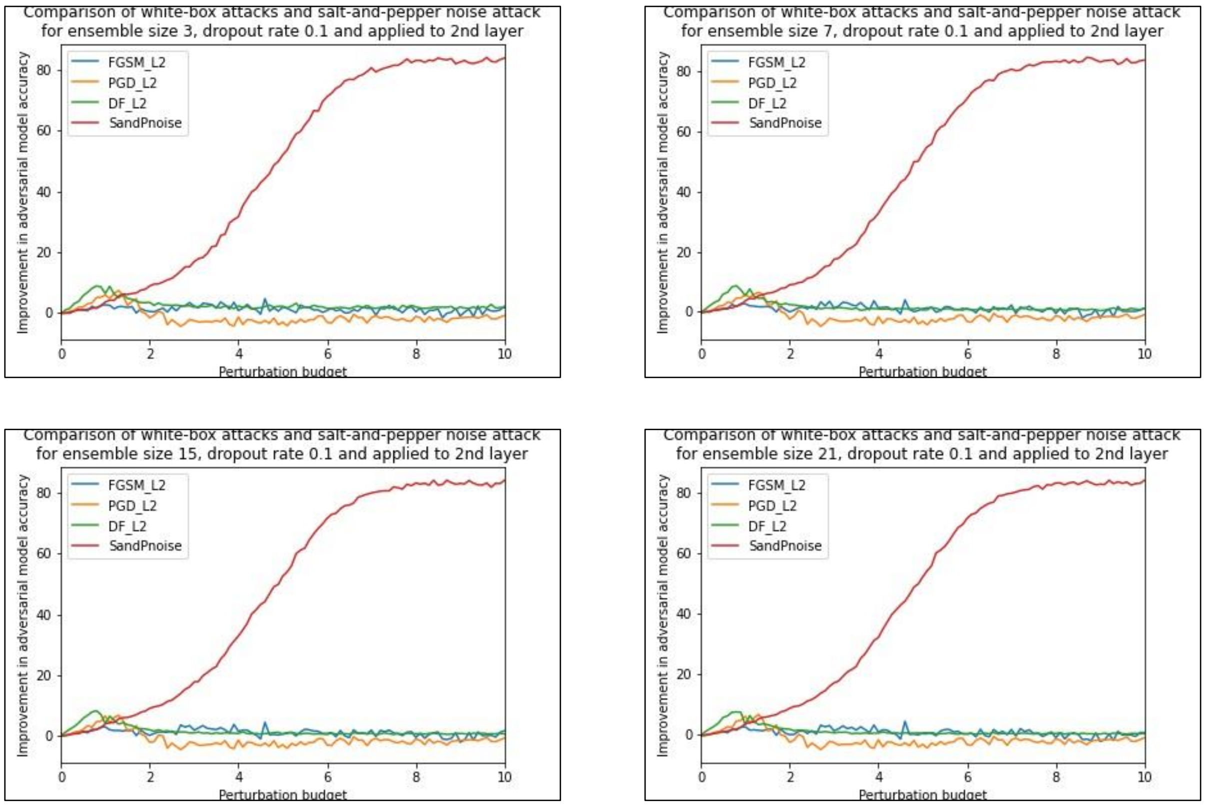


Figure 17: Plot of comparison between white-box attacks and salt-and-pepper noise attack in improvement in average adversarial model accuracy against perturbation budget

4.1.7. Presence of noise in graphs

A noticeable characteristic of the graphs of adversarial model accuracy for when test-time dropout is applied is the noise that is present, as seen in figure 18. Technically speaking, an adversarial model accuracy curve should be strictly decreasing with perturbation budget, as once an adversarial example has been found for an input in the sample for a given perturbation budget, this input is also misclassified for any higher perturbation budgets. However, as seen in figure 18, there are points where the adversarial model accuracy increases with increasing perturbation budget. This is because when test-time dropout is applied, the classifier is no longer deterministic, even with the application of ensembling. As a result, the curve does not have to be strictly decreasing, as an adversarial example that was misclassified by the ensemble classifier on one occasion may be classified correctly on the next, leading to the adversarial model accuracy increasing with perturbation budget.

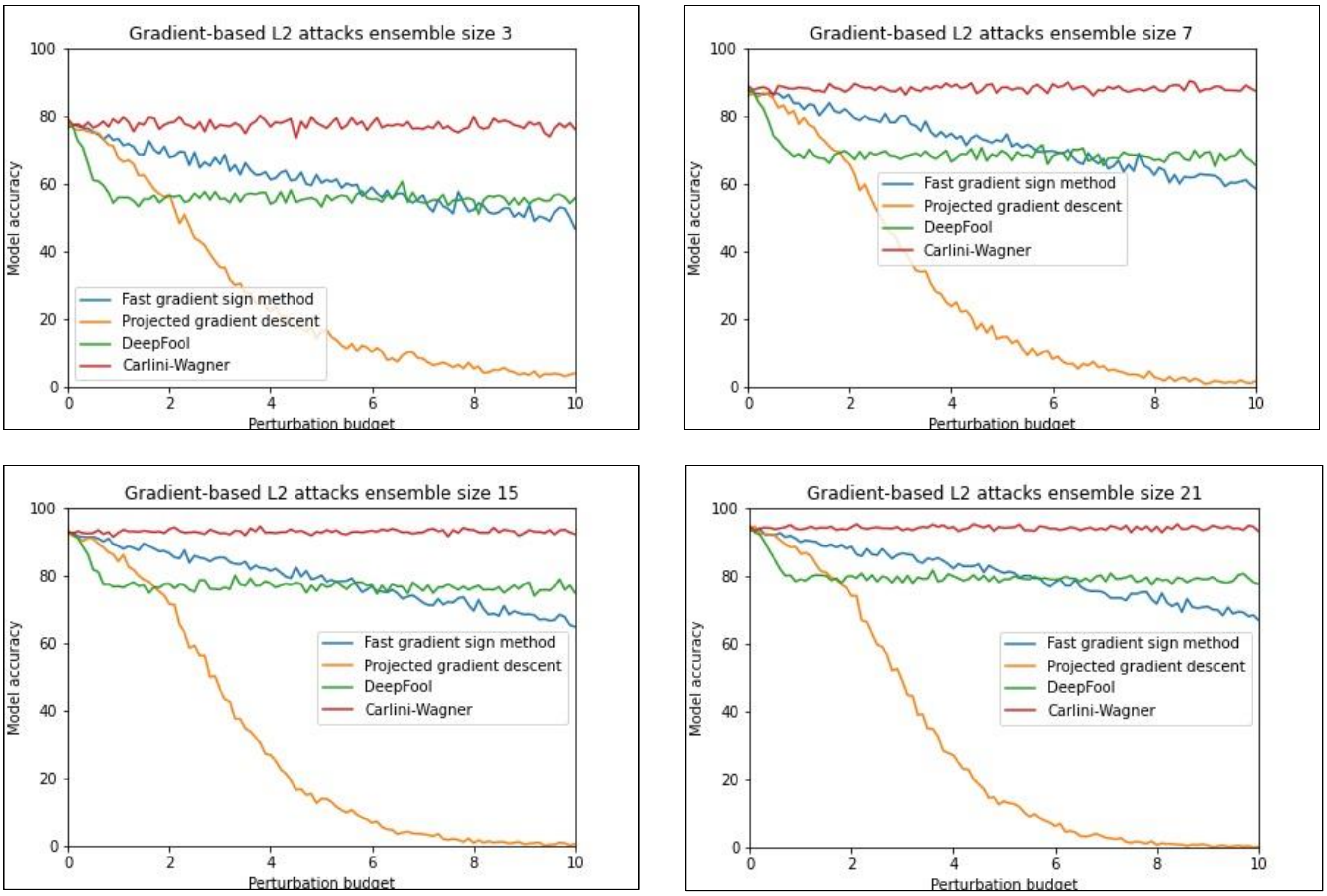


Figure 18: Plot of improvement in adversarial accuracy against perturbation budget. Unlike against the undefended underlying victim model, there is a lot of noise in the results curves due to non-determinism

4.2. Generalisation performance

The results of the impact of the application of test-time dropout on generalisation performance can be seen in appendix 9. For dropout rates between 0.1 and 0.7, we observe that the generalisation performance of the classifier surprisingly does not decrease significantly, if at all, with an increase in the dropout rate. However, for a dropout rate of 0.9, generalisation performance decreases significantly, but increased ensemble size, as expected, can compensate for this loss of accuracy.

5. Conclusions and further work

5.1. Conclusion

On the balance of the evidence and arguments presented in section 4, the author believes it is fair to conclude that test-time dropout when applied at an activation map level does not improve adversarial robustness, and that even in cases where it does seem to, it is possible that it is not through the gradient masking mechanism theorised in section 2.6.2.

With regards to why this might be, recall from section 2.4.4 that the success of a randomisation-based scheme in improving adversarial robustness is dependent on the diversity of the adversarial subspaces of the possible models that can be created by the randomness. It is possible that in the current experimental setup, the diversity of the possible models caused by implementing test-time dropout is not high enough to cause significant gradient masking. Whilst the dropout rate could not have been varied more than it has been in the current experimental setup, the combination of layers which test-time dropout was applied to could have been. For example, test-time dropout could have been applied to multiple layers or to different types of layers simultaneously. Therefore, it's possible that this lack of variety in the application of test-time dropout in the present work is what caused the lack of adversarial robustness due to test-time dropout.

5.2. Comparison to other works

When comparing the experimental results of the present work with those of Wang et al [82], similarities are observed. Though most of the results, as discussed above, do not provide evidence that test-time dropout at an activation map level improves adversarial robustness, there is one particular test-time dropout configuration that does. Throughout both the adversarial robustness results and generalisation performance results, the performance of test-time dropout has been superior when applied to the 7th layer, which is the first fully connected layer. This is the same layer which Wang et al [82] applied test-time dropout to, and they too saw good results from it. With two works now showing the effectiveness of applying test-time dropout to the fully connected layers at the end of a CNN, the evidence for the benefit of test-time dropout applied to this layer specifically as a method for improving adversarial robustness whilst maintaining generalisation performance is becoming significant. It is possible that dropout simply works well on conventional fully connected layers and not on convolutional layers.

5.3. Future work

Between the unaddressed knowledge gaps and shortcomings in the experimental setup of both the present work and existing works, there remain many unexplored aspects of the application of test-time dropout for adversarial robustness. The following sections enumerate some of them and discusses what paths future work on the subject might follow.

5.3.1. Victim model setup

1. The only domain in which test-time dropout has been applied is image classification. Future work might attempt to explore the application of the technique in speech recognition or natural language processing.
2. All works using test-time dropout have only used simple datasets, such as MNIST AND CIFAR-10, so any effectiveness found on these may not carry over to other more complex datasets.

3. With regards to the present work, a simple network architecture has been used, so it is unknown how effective test-time dropout at the activation map level would be when applied to a more complex network type such as ResNets. Addressing this question might help to address knowledge gap 1 as well.
4. As well as this, dropout was not applied at test-time to the convolutional layers, although this is anyway not typically done.

5.3.2. Threat model setup

1. Test-time dropout, in the present work nor in any other work, has not been tested against many black-box attacks, such as transfer-based attacks, score-based attacks or even a more sophisticated type of decision-based attack. Future work might aim at assessing its performance against these.
2. In the present work, only untargeted attacks were used by the adversary. Targeted attacks are generally considered stronger, so even in cases when test-time dropout is effective against untargeted attacks, it is unknown whether it would be effective against targeted attacks.
3. In the present work, the adversary does not know that randomisation is being applied. Simply knowing that randomisation is being applied, even without knowing the details of its application, would allow the adversary to use a counterattack strategy that might help such as EoT, which is mentioned and explained in section 2.4.4. Future work might explore whether EoT is as effective against test-time dropout at an activation map level as it is against other randomisation-based schemes.

5.3.3. Defensive strategy and configurations setup

1. In the present work, test-time dropout has only been assessed by applying it to one layer at a time. As mentioned in section 5.1, it's possible that doing this limits the diversity of the possible models, so future work might try applying test-time dropout to multiple layers at a time. If performance improves, it might be due to an increase in diversity of the possible models.
2. Only relatively small ensemble sizes are reviewed in the current work; other works have been known to use ensemble sizes of 50-100. Future work might apply ensemble sizes in this range.
3. In the present work, ensembling is only used on final testing queries and not on generation queries, which is obviously not possible to implement in reality, where it must be applied on all queries or on none at all. However, the solution is not as simple as just applying ensembling on every query. This is because in doing so, when the adversary infers the gradient from the ensemble, it will receive the average of the gradients from each of the forward passes used. This would be like what the EoT method does, which has been found to be effective as a counterattack strategy against randomisation-based schemes. Therefore, simply applying ensembling on every query might defeat the purpose of the randomisation-based scheme. However, it is possible that an ensemble size exists which is small enough that the gradients received from it do not point towards the common adversarial subspace of all the models, but large enough that generalisation performance issues are mitigated. Future work, therefore, might build on the present work by applying ensembling on every query and finding the optimal ensemble size for ensuring that doing so doesn't mitigate the defensive effects of test-time dropout as a randomisation-based scheme.

5.3.4. Unaddressed knowledge gaps

1. With regards to knowledge gap 2, we have already discussed how applying test-time dropout to multiple layers might increase the diversity of the possible models. However, no work has been done on how this diversity can be encouraged during the training of the underlying victim model, so future work might look at potential regularisation terms that can be included in the objective function of a training algorithm for the underlying victim model for encouraging this diversity.
2. As discussed in section 2.5, the technology for assessing the objective adversarial robustness of a DNN is still some time away from being applicable on a wide scale, so it is unlikely that knowledge gap 3 will be addressed soon. As such, it should be considered as the lowest priority potential path for future work.

6. Legal, social, ethical, and professional issues

The four principles of the British Computer society (BCS) Code of Conduct have been followed and applied throughout the development of the present work, as have the Rules of Conduct of the Institution of Engineering and Technology (IET).

References

[1] K. I. Qazi, H. K. Lam, B. Xiao, G. Ouyang, and X. Yin, ‘Classification of epilepsy using computational intelligence techniques’, *CAAI Transactions on Intelligence Technology*, vol. 1, no. 2, pp. 137–149, Apr. 2016, doi: 10.1016/j.trit.2016.08.001.

[2] S. A. Hussain and A. S. A. A. Balushi, ‘A real time face emotion classification and recognition using deep learning model’, *J. Phys.: Conf. Ser.*, vol. 1432, p. 012087, Jan. 2020, doi: 10.1088/1742-6596/1432/1/012087.

[3] W. R. L. da Silva and D. S. de Lucena, ‘Concrete Cracks Detection Based on Deep Learning Image Classification’, *Proceedings*, vol. 2, no. 8, Art. no. 8, 2018, doi: 10.3390/ICEM18-05387.

[4] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, ‘Flexible, High Performance Convolutional Neural Networks for Image Classification’, p. 6.

[5] M. Pritt and G. Chern, ‘Satellite Image Classification with Deep Learning’, in *2017 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, Oct. 2017, pp. 1–7. doi: 10.1109/AIPR.2017.8457969.

[6] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan, ‘Speech Recognition Using Deep Neural Networks: A Systematic Review’, *IEEE Access*, vol. 7, pp. 19143–19165, 2019, doi: 10.1109/ACCESS.2019.2896880.

[7] D. A. Reynolds, ‘An overview of automatic speaker recognition technology’, in *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, May 2002, vol. 4, p. IV-4072-IV–4075. doi: 10.1109/ICASSP.2002.5745552.

[8] R. A. Khalil, E. Jones, M. I. Babar, T. Jan, M. H. Zafar, and T. Alhussain, ‘Speech Emotion Recognition Using Deep Learning Techniques: A Review’, *IEEE Access*, vol. 7, pp. 117327–117345, 2019, doi: 10.1109/ACCESS.2019.2936124.

[9] R. Zhao, R. Yan, Z. Chen, K. Mao, P. Wang, and R. X. Gao, ‘Deep learning and its applications to machine health monitoring’, *Mechanical Systems and Signal Processing*, vol. 115, pp. 213–237, Jan. 2019, doi: 10.1016/j.ymssp.2018.05.050.

[10] D. W. Otter, J. R. Medina, and J. K. Kalita, ‘A Survey of the Usages of Deep Learning for Natural Language Processing’, *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 2, pp. 604–624, Feb. 2021, doi: 10.1109/TNNLS.2020.2979670.

[11] E. Batbaatar, M. Li, and K. H. Ryu, ‘Semantic-Emotion Neural Network for Emotion Recognition From Text’, *IEEE Access*, vol. 7, pp. 111866–111878, 2019, doi: 10.1109/ACCESS.2019.2934529.

[12] D. Silver *et al.*, ‘Mastering the game of Go with deep neural networks and tree search’, *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961.

[13] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, ‘Deep learning applications and challenges in big data analytics’, *Journal of Big Data*, vol. 2, no. 1, p. 1, Feb. 2015, doi: 10.1186/s40537-014-0007-7.

[14] T. Wang *et al.*, ‘A review on medical imaging synthesis using deep learning and its clinical applications’, *Journal of Applied Clinical Medical Physics*, vol. 22, no. 1, pp. 11–36, 2021, doi: 10.1002/acm2.13121.

[15] N. Aloysius and M. Geetha, ‘A review on deep convolutional neural networks’, in *2017 International Conference on Communication and Signal Processing (ICCSP)*, Apr. 2017, pp. 0588–0592. doi: 10.1109/ICCSP.2017.8286426.

[16] S. Saha, ‘A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way’, *Medium*, Dec. 17, 2018. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (accessed Aug. 22, 2021).

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, ‘Dropout: A Simple Way to Prevent Neural Networks from Overfitting’, *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[18] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, ‘Improving neural networks by preventing co-adaptation of feature detectors’, *arXiv:1207.0580 [cs]*, Jul. 2012, Accessed: Jun. 10, 2021. [Online]. Available: http://arxiv.org/abs/1207.0580

[19] A. ben khalifa and H. Frigui, ‘Multiple Instance Fuzzy Inference Neural Networks’, Oct. 2016.
[20] A. Geron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2nd New edition. Beijing China ; Sebastopol, CA: OReilly, 2019.
[21] L. Breiman, ‘Random Forests’, *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001, doi: 10.1023/A:1010933404324.
[22] C. Szegedy *et al.*, ‘Intriguing properties of neural networks’, *arXiv:1312.6199 [cs]*, Feb. 2014, Accessed: Jun. 09, 2021. [Online]. Available: http://arxiv.org/abs/1312.6199
[23] I. J. Goodfellow, J. Shlens, and C. Szegedy, ‘Explaining and Harnessing Adversarial Examples’, *arXiv:1412.6572 [cs, stat]*, Mar. 2015, Accessed: Jun. 07, 2021. [Online]. Available: http://arxiv.org/abs/1412.6572
[24] A. Fawzi, S.-M. Moosavi-Dezfooli, and P. Frossard, ‘Robustness of classifiers: from adversarial to random noise’, *arXiv:1608.08967 [cs, stat]*, Aug. 2016, Accessed: Jun. 25, 2021. [Online]. Available: http://arxiv.org/abs/1608.08967
[25] T. Tanay and L. Griffin, ‘A Boundary Tilting Persepective on the Phenomenon of Adversarial Examples’, *arXiv:1608.07690 [cs, stat]*, Aug. 2016, Accessed: Jun. 20, 2021. [Online]. Available: http://arxiv.org/abs/1608.07690
[26] A. Shafahi, W. R. Huang, C. Studer, S. Feizi, and T. Goldstein, ‘Are adversarial examples inevitable?’, *arXiv:1809.02104 [cs, stat]*, Feb. 2020, Accessed: Jun. 25, 2021. [Online]. Available: http://arxiv.org/abs/1809.02104
[27] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, P. Frossard, and S. Soatto, ‘Robustness of classifiers to universal perturbations: a geometric perspective’, *arXiv:1705.09554 [cs, stat]*, Mar. 2021, Accessed: Jun. 25, 2021. [Online]. Available: http://arxiv.org/abs/1705.09554
[28] A. Fawzi, O. Fawzi, and P. Frossard, ‘Analysis of classifiers’ robustness to adversarial perturbations’, *arXiv:1502.02590 [cs, stat]*, Mar. 2016, Accessed: Jun. 22, 2021. [Online]. Available: http://arxiv.org/abs/1502.02590
[29] S. Bubeck, E. Price, and I. Razenshteyn, ‘Adversarial examples from computational constraints’, *arXiv:1805.10204 [cs, stat]*, May 2018, Accessed: Jun. 25, 2021. [Online]. Available: http://arxiv.org/abs/1805.10204
[30] K. Ren, T. Zheng, Z. Qin, and X. Liu, ‘Adversarial Attacks and Defenses in Deep Learning’, *Engineering*, vol. 6, no. 3, pp. 346–360, Mar. 2020, doi: 10.1016/j.eng.2019.12.012.
[31] H. Xu *et al.*, ‘Adversarial Attacks and Defenses in Images, Graphs and Text: A Review’, *Int. J. Autom. Comput.*, vol. 17, no. 2, pp. 151–178, Apr. 2020, doi: 10.1007/s11633-019-1211-x.
[32] X. Yuan, P. He, Q. Zhu, and X. Li, ‘Adversarial Examples: Attacks and Defenses for Deep Learning’, *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 9, pp. 2805–2824, Sep. 2019, doi: 10.1109/TNNLS.2018.2886017.
[33] N. Akhtar and A. Mian, ‘Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey’, *IEEE Access*, vol. 6, pp. 14410–14430, 2018, doi: 10.1109/ACCESS.2018.2807385.
[34] A. Goel, ‘An Empirical Review of Adversarial Defenses’, *arXiv:2012.06332 [cs]*, Dec. 2020, Accessed: Apr. 26, 2021. [Online]. Available: http://arxiv.org/abs/2012.06332
[35] N. Carlini *et al.*, ‘On Evaluating Adversarial Robustness’, *arXiv:1902.06705 [cs, stat]*, Feb. 2019, Accessed: Jun. 14, 2021. [Online]. Available: http://arxiv.org/abs/1902.06705
[36] A. Kurakin, I. Goodfellow, and S. Bengio, ‘Adversarial examples in the physical world’, *arXiv:1607.02533 [cs, stat]*, Feb. 2017, Accessed: Jun. 30, 2021. [Online]. Available: http://arxiv.org/abs/1607.02533
[37] N. Carlini and D. Wagner, ‘Towards Evaluating the Robustness of Neural Networks’, in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 39–57. doi: 10.1109/SP.2017.49.
[38] D. P. Kingma and J. Ba, ‘Adam: A Method for Stochastic Optimization’, *arXiv:1412.6980 [cs]*, Jan. 2017, Accessed: Jul. 01, 2021. [Online]. Available: http://arxiv.org/abs/1412.6980
[39] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, ‘DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks’, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 2574–2582. doi: 10.1109/CVPR.2016.282.

[40] T. Zheng, C. Chen, and K. Ren, ‘Distributionally Adversarial Attack’, *arXiv:1808.05537 [cs, stat]*, Dec. 2018, Accessed: Jul. 02, 2021. [Online]. Available: http://arxiv.org/abs/1808.05537

[41] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, ‘Generating Adversarial Examples with Adversarial Networks’, *arXiv:1801.02610 [cs, stat]*, Feb. 2019, Accessed: Jul. 02, 2021. [Online]. Available: http://arxiv.org/abs/1801.02610

[42] C. Xiao, J.-Y. Zhu, B. Li, W. He, M. Liu, and D. Song, ‘Spatially Transformed Adversarial Examples’, *arXiv:1801.02612 [cs, stat]*, Jan. 2018, Accessed: Jul. 01, 2021. [Online]. Available: http://arxiv.org/abs/1801.02612

[43] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, ‘ZOO: Zeroth Order Optimization based Black-box Attacks to Deep Neural Networks without Training Substitute Models’, *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pp. 15–26, Nov. 2017, doi: 10.1145/3128572.3140448.

[44] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, ‘Black-box Adversarial Attacks with Limited Queries and Information’, *arXiv:1804.08598 [cs, stat]*, Jul. 2018, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/1804.08598

[45] Y. Li, L. Li, L. Wang, T. Zhang, and B. Gong, ‘NATTACK: Learning the Distributions of Adversarial Examples for an Improved Black-Box Attack on Deep Neural Networks’, *arXiv:1905.00441 [cs, stat]*, Dec. 2019, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/1905.00441

[46] W. Brendel, J. Rauber, and M. Bethge, ‘Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models’, *arXiv:1712.04248 [cs, stat]*, Feb. 2018, Accessed: Aug. 18, 2021. [Online]. Available: http://arxiv.org/abs/1712.04248

[47] H. Li, X. Xu, X. Zhang, S. Yang, and B. Li, ‘QEBA: Query-Efficient Boundary-Based Blackbox Attack’, *arXiv:2005.14137 [cs, stat]*, May 2020, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/2005.14137

[48] J. Chen, M. I. Jordan, and M. J. Wainwright, ‘HopSkipJumpAttack: A Query-Efficient Decision-Based Attack’, *arXiv:1904.02144 [cs, math, stat]*, Apr. 2020, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/1904.02144

[49] J. Rauber, W. Brendel, and M. Bethge, ‘Foolbox: A Python toolbox to benchmark the robustness of machine learning models’, *arXiv:1707.04131 [cs, stat]*, Mar. 2018, Accessed: Jul. 13, 2021. [Online]. Available: http://arxiv.org/abs/1707.04131

[50] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, ‘Towards Deep Learning Models Resistant to Adversarial Attacks’, *arXiv:1706.06083 [cs, stat]*, Sep. 2019, Accessed: Aug. 10, 2021. [Online]. Available: http://arxiv.org/abs/1706.06083

[51] H. Kannan, A. Kurakin, and I. Goodfellow, ‘Adversarial Logit Pairing’, *arXiv:1803.06373 [cs, stat]*, Mar. 2018, Accessed: Aug. 10, 2021. [Online]. Available: http://arxiv.org/abs/1803.06373

[52] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, ‘Ensemble Adversarial Training: Attacks and Defenses’, *arXiv:1705.07204 [cs, stat]*, Apr. 2020, Accessed: Aug. 10, 2021. [Online]. Available: http://arxiv.org/abs/1705.07204

[53] M. Hein and M. Andriushchenko, ‘Formal Guarantees on the Robustness of a Classifier against Adversarial Manipulation’, *arXiv:1705.08475 [cs, stat]*, Nov. 2017, Accessed: Jul. 05, 2021. [Online]. Available: http://arxiv.org/abs/1705.08475

[54] A. Raghunathan, J. Steinhardt, and P. Liang, ‘Certified Defenses against Adversarial Examples’, *arXiv:1801.09344 [cs]*, Oct. 2020, Accessed: Aug. 12, 2021. [Online]. Available: http://arxiv.org/abs/1801.09344

[55] A. Raghunathan, J. Steinhardt, and P. Liang, ‘Semidefinite relaxations for certifying robustness to adversarial examples’, *arXiv:1811.01057 [cs, stat]*, Nov. 2018, Accessed: Aug. 12, 2021. [Online]. Available: http://arxiv.org/abs/1811.01057

[56] E. Wong, F. Schmidt, J. H. Metzen, and J. Z. Kolter, ‘Scaling provable adversarial defenses’, in *Advances in Neural Information Processing Systems*, 2018, vol. 31. Accessed: Aug. 12, 2021. [Online]. Available: https://papers.nips.cc/paper/2018/hash/358f9e7be09177c17d0d17ff73584307-Abstract.html

[57] S. Kariyappa and M. K. Qureshi, 'Improving Adversarial Robustness of Ensembles with Diversity Training', *arXiv:1901.09981 [cs, stat]*, Jan. 2019, Accessed: Jul. 18, 2021. [Online]. Available: http://arxiv.org/abs/1901.09981
[58] T. Pang, K. Xu, C. Du, N. Chen, and J. Zhu, 'Improving Adversarial Robustness via Promoting Ensemble Diversity', in *International Conference on Machine Learning*, May 2019, pp. 4970–4979. Accessed: Jul. 18, 2021. [Online]. Available: http://proceedings.mlr.press/v97/pang19a.html
[59] W. Xu, D. Evans, and Y. Qi, 'Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks', *Proceedings 2018 Network and Distributed System Security Symposium*, 2018, doi: 10.14722/ndss.2018.23198.
[60] D. Meng and H. Chen, 'MagNet: a Two-Pronged Defense against Adversarial Examples', *arXiv:1705.09064 [cs]*, Sep. 2017, Accessed: Aug. 12, 2021. [Online]. Available: http://arxiv.org/abs/1705.09064
[61] F. Liao, M. Liang, Y. Dong, T. Pang, X. Hu, and J. Zhu, 'Defense against Adversarial Attacks Using High-Level Representation Guided Denoiser', *arXiv:1712.02976 [cs]*, May 2018, Accessed: Aug. 12, 2021. [Online]. Available: http://arxiv.org/abs/1712.02976
[62] G. Hinton, O. Vinyals, and J. Dean, 'Distilling the Knowledge in a Neural Network', *arXiv:1503.02531 [cs, stat]*, Mar. 2015, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/1503.02531
[63] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, 'Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks', *arXiv:1511.04508 [cs, stat]*, Mar. 2016, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/1511.04508
[64] J. Buckman, A. Roy, C. Raffel, and I. Goodfellow, 'THERMOMETER ENCODING: ONE HOT WAY TO RESIST ADVERSARIAL EXAMPLES', p. 22, 2018.
[65] A. Athalye, N. Carlini, and D. Wagner, 'Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples', *arXiv:1802.00420 [cs]*, Jul. 2018, Accessed: Aug. 12, 2021. [Online]. Available: http://arxiv.org/abs/1802.00420
[66] X. Liu, M. Cheng, H. Zhang, and C.-J. Hsieh, 'Towards Robust Neural Networks via Random Self-ensemble', *arXiv:1712.00673 [cs, stat]*, Jul. 2018, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/1712.00673
[67] C. Xie, J. Wang, Z. Zhang, Z. Ren, and A. Yuille, 'Mitigating Adversarial Effects Through Randomization', *arXiv:1711.01991 [cs]*, Feb. 2018, Accessed: Aug. 10, 2021. [Online]. Available: http://arxiv.org/abs/1711.01991
[68] N. Carlini, G. Katz, C. Barrett, and D. L. Dill, 'Provably Minimally-Distorted Adversarial Examples', *arXiv:1709.10207 [cs]*, Feb. 2018, Accessed: Aug. 13, 2021. [Online]. Available: http://arxiv.org/abs/1709.10207
[69] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, 'Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks', *arXiv:1702.01135 [cs]*, May 2017, Accessed: Aug. 22, 2021. [Online]. Available: http://arxiv.org/abs/1702.01135
[70] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, 'Measuring Neural Net Robustness with Constraints', *arXiv:1605.07262 [cs]*, Jun. 2017, Accessed: Jun. 14, 2021. [Online]. Available: http://arxiv.org/abs/1605.07262
[71] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, 'Safety Verification of Deep Neural Networks', *arXiv:1610.06940 [cs, stat]*, May 2017, Accessed: Jun. 14, 2021. [Online]. Available: http://arxiv.org/abs/1610.06940
[72] S. Ahamed, 'Estimating uncertainty of earthquake rupture using Bayesian neural network', *arXiv:1911.09660 [physics, stat]*, Nov. 2019, Accessed: Aug. 29, 2021. [Online]. Available: http://arxiv.org/abs/1911.09660
[73] Y. Li and Y. Gal, 'Dropout Inference in Bayesian Neural Networks with Alpha-divergences', *arXiv:1703.02914 [cs, stat]*, Mar. 2017, Accessed: Jun. 29, 2021. [Online]. Available: http://arxiv.org/abs/1703.02914

[74] Y. Gal and Z. Ghahramani, 'Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning', *arXiv:1506.02142 [cs, stat]*, Oct. 2016, Accessed: Jun. 10, 2021. [Online]. Available: http://arxiv.org/abs/1506.02142

[75] A. Labach, H. Salehinejad, and S. Valaee, 'Survey of Dropout Methods for Deep Neural Networks', *arXiv:1904.13310 [cs]*, Oct. 2019, Accessed: Jun. 10, 2021. [Online]. Available: http://arxiv.org/abs/1904.13310

[76] I. Cortés-Ciriano and A. Bender, 'Reliable Prediction Errors for Deep Neural Networks Using Test-Time Dropout', *J. Chem. Inf. Model.*, vol. 59, no. 7, pp. 3330–3339, Jul. 2019, doi: 10.1021/acs.jcim.9b00297.

[77] O. Dürr, E. Murina, D. Siegismund, V. Tolkachev, S. Steigele, and B. Sick, 'Know When You Don't Know: A Robust Deep Learning Approach in the Presence of Unknown Phenotypes', *ASSAY and Drug Development Technologies*, vol. 16, no. 6, pp. 343–349, Aug. 2018, doi: 10.1089/adt.2018.859.

[78] K. Brach, B. Sick, and O. Dürr, 'Single Shot MC Dropout Approximation', *arXiv:2007.03293 [cs, stat]*, Jul. 2020, Accessed: Jun. 10, 2021. [Online]. Available: http://arxiv.org/abs/2007.03293

[79] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, 'Detecting Adversarial Samples from Artifacts', *arXiv:1703.00410 [cs, stat]*, Nov. 2017, Accessed: Jul. 13, 2021. [Online]. Available: http://arxiv.org/abs/1703.00410

[80] G. S. Dhillon *et al.*, 'Stochastic Activation Pruning for Robust Adversarial Defense', *arXiv:1803.01442 [cs, stat]*, Mar. 2018, Accessed: Jun. 29, 2021. [Online]. Available: http://arxiv.org/abs/1803.01442

[81] G. S. Dhillon and N. Carlini, 'Erratum Concerning the Obfuscated Gradients Attack on Stochastic Activation Pruning', *arXiv:2010.00071 [cs, stat]*, Sep. 2020, Accessed: Aug. 17, 2021. [Online]. Available: http://arxiv.org/abs/2010.00071

[82] S. Wang *et al.*, 'Defensive Dropout for Hardening Deep Neural Networks under Adversarial Attacks', *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–8, Nov. 2018, doi: 10.1145/3240765.3264699.

[83] 'MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges'. http://yann.lecun.com/exdb/mnist/ (accessed Aug. 22, 2021).

[84] 'CIFAR-10 and CIFAR-100 datasets'. https://www.cs.toronto.edu/~kriz/cifar.html (accessed Aug. 22, 2021).

[85] 'Welcome to Python.org', *Python.org*. https://www.python.org/ (accessed Aug. 23, 2021).

[86] A. Paszke *et al.*, 'PyTorch: An Imperative Style, High-Performance Deep Learning Library', *arXiv:1912.01703 [cs, stat]*, Dec. 2019, Accessed: Aug. 23, 2021. [Online]. Available: http://arxiv.org/abs/1912.01703

[87] J. Rauber, W. Brendel, and M. Bethge, 'Foolbox: A Python toolbox to benchmark the robustness of machine learning models', *arXiv:1707.04131 [cs, stat]*, Mar. 2018, Accessed: Aug. 27, 2021. [Online]. Available: http://arxiv.org/abs/1707.04131

[88] 'Google Colab'. https://colab.research.google.com/signup (accessed Aug. 23, 2021).

[89] K. He, X. Zhang, S. Ren, and J. Sun, 'Deep Residual Learning for Image Recognition', *arXiv:1512.03385 [cs]*, Dec. 2015, Accessed: Aug. 23, 2021. [Online]. Available: http://arxiv.org/abs/1512.03385

Appendices

Appendix 1: Percentage improvement in average adversarial model accuracy against the FGSM L_2 generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.59	1.17	1.11	-0.09	0.29	0.36	1.08	-0.41
0.3	-3.2	3.86	3.8	-0.75	1.71	1.63	4.87	-0.2
0.5	-8.81	2.74	2.68	-4.31	1.45	1.5	9.41	-0.27
0.7	-17.98	-10.08	-9.96	-13.9	-3.58	-3.42	13.46	0.34
0.9	-33.6	-42.3	-42.19	-34.89	-26.65	-26.54	7.44	-14.34

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.73	1.05	1.06	0.05	0.36	0.4	1.12	-0.39
0.3	-2.88	5.07	5.02	-0.04	2.42	2.42	4.92	-0.11
0.5	-7.47	7.71	7.71	-2.35	4.34	4.24	9.48	-0.17
0.7	-14.51	1.23	1.27	-9.45	4.18	4.21	13.97	2.19
0.9	-27.04	-39.22	-39.03	-29.42	-14.75	-14.92	13.88	-3.89

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.8	0.92	0.94	0.02	0.4	0.37	1.15	-0.37
0.3	-2.76	5.56	5.46	0.17	2.66	2.59	4.93	-0.07
0.5	-7.1	9.73	9.72	-1.72	5.34	5.25	9.45	-0.16
0.7	-13.99	7.7	7.61	-7.87	7.46	7.44	14.13	2.51
0.9	-22.79	-37.82	-37.59	-28	-8.24	-8.42	16.02	0.1

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.8	0.84	0.86	0.02	0.4	0.37	1.16	-0.37
0.3	-2.79	5.67	5.48	0.2	2.72	2.7	4.96	-0.06
0.5	-7.09	10.2	10.16	-1.62	5.51	5.52	9.49	-0.14
0.7	-13.95	9.43	9.32	-7.39	8.39	8.39	14.15	2.59
0.9	-20.84	-37.42	-37.44	-27.76	-6.51	-6.36	16.4	0.89

Appendix 2: Percentage improvement in average adversarial model accuracy against the PGD L_2 generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.74	-1.22	-1.2	-1.72	-1.43	-1.37	0.91	-0.26
0.3	-3	-2.55	-2.52	-3.01	-2.62	-2.5	4.2	-1.57
0.5	-4.33	-1.11	-1.11	-3.68	-3.61	-3.59	8.99	-1.59
0.7	-4.12	1.61	1.46	-2.68	-2.71	-2.82	14.57	-3.75
0.9	-8.21	-7.95	-7.8	-4.57	-1.79	-1.74	16.53	2.84

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.89	-1.29	-1.29	-1.71	-1.46	-1.36	0.89	-0.26
0.3	-3.04	-2.64	-2.59	-2.98	-2.56	-2.44	4.18	-1.54
0.5	-4.09	-0.7	-0.83	-3.54	-3.24	-3.27	8.98	-1.59
0.7	-2.65	3.82	3.86	-1.71	-1.56	-1.6	14.59	-3.35
0.9	-6.69	-7.33	-7.3	-2.83	1.68	1.56	19.41	6.78

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.96	-1.35	-1.36	-1.71	-1.48	-1.38	0.88	-0.26
0.3	-3.03	-2.68	-2.55	-2.96	-2.56	-2.39	4.18	-1.54
0.5	-4.04	-0.66	-0.75	-3.42	-3.17	-3.15	8.97	-1.56
0.7	-2.19	5.16	5.26	-1.38	-1.08	-1.14	14.6	-3.38
0.9	-5.56	-7.28	-7.22	-2.14	3.53	3.6	20.42	8.43

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.97	-1.36	-1.36	-1.72	-1.48	-1.39	0.88	-0.26
0.3	-3.04	-2.7	-2.59	-2.94	-2.54	-2.39	4.18	-1.52
0.5	-4.02	-0.66	-0.69	-3.42	-3.15	-3.15	8.97	-1.57
0.7	-2.06	5.66	5.64	-1.26	-0.97	-1.03	14.6	-3.4
0.9	-5.07	-7.39	-7.37	-2	4.1	4.12	20.64	8.71

Appendix 3: Percentage improvement in average adversarial model accuracy against the DeepFool L_2 generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	2.01	2.5	2.28	1.3	1.58	2.25	1.4	1.65
0.3	3.04	8.23	7.9	3.08	4.31	4.05	1.56	1.53
0.5	10.48	31.31	30.87	10.11	13.52	13.15	1.85	1.5
0.7	28.31	47.42	48.76	30.39	32.57	33.9	2.34	7.48
0.9	29.1	20.23	20.43	34.67	35.31	35.71	17.03	21.19

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	1.08	1.68	1.55	0.9	1	1.76	1.33	1.68
0.3	1.94	6.5	5.93	2.06	2.8	2.9	1.4	1.5
0.5	9.91	34.66	34.3	7.66	11.72	11.35	1.66	1.49
0.7	30.45	59.95	61.7	31.66	37.46	38.39	2.08	7.7
0.9	36.44	23.97	24.49	43.88	46.93	47.38	18.28	26.57

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	0.81	1.35	1.26	0.77	0.9	1.65	1.29	1.68
0.3	1.3	5.54	5	1.78	2.28	2.5	1.36	1.49
0.5	9.62	36.32	35.64	6.62	10.44	10.06	1.59	1.5
0.7	30.03	68.26	69.84	31.38	39.49	40.68	1.98	7.67
0.9	42.25	25.82	26.36	48.83	52.78	53.55	18.79	28.91

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	0.76	1.27	1.19	0.77	0.89	1.63	1.28	1.68
0.3	1.13	5.24	4.75	1.65	2.17	2.45	1.34	1.51
0.5	9.51	36.86	36.03	6.25	10	9.66	1.59	1.47
0.7	29.84	70.63	72.27	31.19	40.09	41.33	1.91	7.66
0.9	44.7	26.04	26.92	50.12	54.45	55.24	18.97	29.56

Appendix 4: Percentage improvement in average adversarial model accuracy against the Carlini-Wagner L_2 generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	90.99	90.55	90.64	91.21	91.17	91.33	88.82	90.62
0.3	91.64	90.9	90.98	91.48	91.22	91.2	89.76	91.1
0.5	90.73	87.6	87.62	90.24	89.79	89.83	91.54	91.06
0.7	83.78	70.59	70.49	82.75	83.67	83.61	91.47	89.45
0.9	45.3	25.76	25.78	45.28	48.6	48.85	80.94	62.68

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	91.16	90.78	90.85	91.44	91.52	91.55	89.89	91.22
0.3	91.76	91.36	91.48	91.76	91.57	91.54	90.31	91.47
0.5	91.61	90.45	90.48	91.32	91.09	91.11	91.74	91.37
0.7	89.86	81.3	81	88.24	89.2	89.14	91.81	91.4
0.9	56.66	29.07	28.96	56.17	62.92	63.05	88.72	75.37

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	91.23	90.84	90.94	91.53	91.68	91.63	90.29	91.44
0.3	91.75	91.46	91.57	91.84	91.7	91.7	90.59	91.67
0.5	91.76	91.09	91.11	91.67	91.43	91.46	91.81	91.47
0.7	91.2	86.07	85.99	90.08	90.59	90.57	91.9	91.64
0.9	65.99	30.4	30.62	61.82	70.55	70.58	91.16	79.98

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	91.24	90.85	90.97	91.55	91.71	91.66	90.42	91.47
0.3	91.75	91.5	91.59	91.86	91.73	91.72	90.63	91.72
0.5	91.79	91.2	91.23	91.75	91.49	91.51	91.81	91.48
0.7	91.43	87.24	87.19	90.38	90.86	90.84	91.91	91.68
0.9	70.05	30.75	30.81	63.32	72.64	72.65	91.57	80.84

Appendix 5: Percentage improvement in average adversarial model accuracy against the FGSM L_∞ generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.53	0.43	0.5	-0.42	0.35	0.41	1.32	-0.14
0.3	-4.45	0.87	0.61	-3.3	-0.05	-0.07	3.52	-0.76
0.5	-9.85	0.86	0.62	-7.22	-1.65	-1.41	5.56	-0.79
0.7	-14.42	-3.05	-3.07	-12.05	-4.37	-4.18	6.64	-1.47
0.9	-13	-16.92	-17.13	-19.24	-11.17	-11.36	8.47	-3.3

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.32	0.31	0.38	-0.35	0.31	0.32	1.39	-0.11
0.3	-4.07	1.41	1.19	-2.97	0.41	0.52	3.62	-0.73
0.5	-9.45	3.36	3.2	-6.85	-0.13	0.1	5.79	-0.78
0.7	-13.77	3.45	3.33	-10.89	-0.32	-0.26	7.08	-0.6
0.9	-8.95	-14.66	-14.65	-17.88	-4.3	-4.47	11.82	3.32

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.25	0.15	0.29	-0.31	0.35	0.3	1.44	-0.1
0.3	-3.87	1.58	1.41	-3	0.44	0.4	3.69	-0.72
0.5	-9.41	4.23	4.15	-6.78	0.27	0.43	5.94	-0.8
0.7	-14.08	7.17	7.18	-10.91	1.27	1.3	7.12	-0.49
0.9	-6.12	-13.69	-13.78	-18.1	-0.29	-0.34	12.72	5.75

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.21	0.11	0.21	-0.3	0.38	0.29	1.42	-0.09
0.3	-3.87	1.56	1.39	-2.99	0.44	0.5	3.75	-0.72
0.5	-9.42	4.39	4.37	-6.77	0.23	0.45	5.88	-0.77
0.7	-14.24	8.16	8.1	-11.08	1.67	1.73	7.11	-0.42
0.9	-4.82	-13.62	-13.67	-18.28	1.02	0.93	12.86	6.2

Appendix 6: Percentage improvement in average adversarial model accuracy against the PGD L_∞ generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.92	-1.38	-1.29	-1.5	-1.57	-1.53	0.45	-0.2
0.3	-1.55	-1.58	-1.54	-2.18	-1.92	-1.92	2.03	-0.84
0.5	-1.28	-0.11	-0.13	-2.13	-1.91	-1.96	4.77	-0.84
0.7	-0.85	2.11	2.2	-1.6	-1.39	-1.34	8.8	-0.93
0.9	-2.88	-1.63	-1.66	-1.85	0.55	0.5	15.42	3.33

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-1	-1.45	-1.33	-1.52	-1.58	-1.56	0.43	-0.19
0.3	-1.74	-1.67	-1.59	-2.16	-1.89	-1.88	2.03	-0.85
0.5	-1.3	0.11	0	-2.01	-1.81	-1.79	4.77	-0.86
0.7	0.34	3.44	3.44	-1.01	-0.68	-0.65	8.82	-0.71
0.9	-2.22	-1.43	-1.4	-0.54	2.08	2.01	17.29	5.39

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-1.04	-1.48	-1.37	-1.53	-1.58	-1.55	0.42	-0.19
0.3	-1.79	-1.72	-1.63	-2.14	-1.9	-1.85	2.02	-0.85
0.5	-1.39	0.06	0.05	-1.98	-1.69	-1.73	4.77	-0.85
0.7	0.58	4.34	4.41	-0.75	-0.39	-0.33	8.82	-0.7
0.9	-1.66	-1.26	-1.31	0.06	2.93	2.86	17.77	6.2

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-1.04	-1.5	-1.38	-1.54	-1.59	-1.55	0.42	-0.19
0.3	-1.79	-1.74	-1.65	-2.14	-1.9	-1.84	2.03	-0.84
0.5	-1.43	0.05	0.03	-2	-1.69	-1.71	4.76	-0.85
0.7	0.6	4.55	4.58	-0.72	-0.31	-0.3	8.82	-0.72
0.9	-1.59	-1.29	-1.31	0.26	3.14	3.2	17.82	6.35

Appendix 7: Percentage improvement in average adversarial model accuracy against the DeepFool L_∞ generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	0.16	0.85	0.68	-0.03	0.49	0.24	-0.05	0.46
0.3	0.25	5.34	5.53	0.76	1.46	1.38	-0.07	0.54
0.5	7.03	25.77	25.46	6.42	8.48	9.36	-0.04	0.69
0.7	26.21	43.07	42.29	24.25	27.97	28.28	0.48	5.71
0.9	25.99	17.83	17.73	31.57	33	33.33	11.84	18.9

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.55	0.16	0.01	-0.27	0.12	-0.03	-0.08	0.46
0.3	-0.42	3.87	4.03	0.03	0.4	0.44	-0.1	0.53
0.5	6.14	28.17	27.63	4.61	6.68	7.77	-0.14	0.66
0.7	27.95	55.48	54.43	24.59	31.67	31.48	0.25	6.09
0.9	32.71	21.69	21.39	40.56	44.59	44.84	12.98	24.43

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.81	-0.09	-0.25	-0.36	0.09	-0.06	-0.09	0.47
0.3	-0.72	3.25	3.47	-0.06	0.15	0.19	-0.14	0.56
0.5	5.73	29.02	28.7	3.86	5.63	6.7	-0.14	0.67
0.7	27.44	63.89	62.6	24.09	33.23	32.36	0.19	6.02
0.9	37.96	23.54	22.9	45.19	50.93	51.04	13.51	26.77

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	-0.84	-0.13	-0.34	-0.36	0.07	-0.09	-0.09	0.47
0.3	-0.77	3.05	3.34	-0.06	0.11	0.13	-0.15	0.55
0.5	5.62	29.39	28.88	3.61	5.29	6.46	-0.15	0.67
0.7	27.02	66.55	65.1	23.77	33.57	32.64	0.2	6.04
0.9	40.15	23.86	23.03	46.34	52.8	52.94	13.52	27.35

Appendix 8: Percentage improvement in average adversarial model accuracy against the salt-and-pepper noise generation method

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	38.74	47.02	45.44	42.96	42.4	42.69	15.94	9.41
0.3	55.14	55.07	55.33	54.28	54.61	54.34	30.37	26.63
0.5	55.22	52.02	52.16	54.87	54.36	54.22	44.77	26.87
0.7	48.2	35.01	35.24	47.03	48.17	48.27	55.81	53.84
0.9	10.07	-9.71	-9.86	9.29	13.28	13.28	45.38	27.31

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	38.24	47.36	45.5	43.65	42.98	43.3	15.66	9.12
0.3	55.44	55.67	55.89	55.14	55.43	55.06	30.61	26.65
0.5	56.14	54.93	55	55.89	55.66	55.57	45.64	26.84
0.7	54.41	45.64	45.84	52.7	53.71	53.74	56.2	55.83
0.9	21.19	-6.18	-6.58	20.5	27.69	27.8	53.33	39.95

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	37.7	47.28	45.19	43.68	43.16	43.46	15.52	9
0.3	55.54	55.84	56.05	55.39	55.7	55.26	30.51	26.61
0.5	56.29	55.6	55.63	56.22	55.96	55.92	46.09	26.89
0.7	55.7	50.63	50.62	54.51	55.1	55.01	56.28	56.11
0.9	30.39	-4.73	-4.96	26.25	35.04	35.18	55.74	44.47

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	37.56	47.21	45.06	43.72	43.21	43.47	15.48	8.96
0.3	55.54	55.87	56.08	55.49	55.77	55.33	30.41	26.59
0.5	56.33	55.72	55.75	56.31	56.01	56.01	46.17	26.91
0.7	55.93	51.73	51.8	54.85	55.34	55.33	56.31	56.16
0.9	34.18	-4.53	-4.62	27.9	37.02	37.35	56.05	45.32

Appendix 9: Generalisation performance (percentage) when test-time dropout is implemented

Ensemble size 3

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	98.6	98.4	98.3	98.6	98.6	98.9	98.8	98.6
0.3	98.6	98.1	97.4	98.6	98	98.1	98.6	98.6
0.5	97.8	94.2	94.6	97.1	96	96.9	98.8	98.6
0.7	91.7	76.8	77.3	90.2	91.1	91.1	97.9	96.7
0.9	53.2	31.9	33.8	49.9	55.9	53.2	88.7	67

Ensemble size 7

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	98.6	98.5	98.4	98.7	98.6	98.7	98.7	98.6
0.3	98.7	98.2	98.4	98.6	98.2	98.4	98.8	98.7
0.5	98.7	97.7	97.6	98.4	97.6	97.9	98.8	98.7
0.7	96.5	87.6	86.6	94.9	96.3	96.5	98.4	98.2
0.9	65.7	35.8	34.5	63	70	67	95.2	80.3

Ensemble size 15

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	98.6	98.6	98.5	98.7	98.7	98.8	98.7	98.7
0.3	98.7	98.7	98.3	98.8	98.3	98.5	98.6	98.6
0.5	98.5	98.3	98.4	98.6	98.3	98.2	98.9	98.6
0.7	97.9	93	92.8	96.9	97.1	97.6	98.6	98.6
0.9	73.5	38.9	36.2	67.7	78.1	75.9	98	86.5

Ensemble size 21

	1stLayer	2ndLayer	3rdLayer	4thLayer	5thLayer	6thLayer	7thLayer	8thLayer
0.1	98.6	98.7	98.6	98.7	98.7	98.7	98.7	98.7
0.3	98.7	98.5	98.3	98.7	98.7	98.6	98.6	98.6
0.5	98.8	98.5	98.3	98.5	98.4	98.3	98.9	98.6
0.7	98.6	93.7	93.6	97.6	97.8	97.1	98.6	98.7
0.9	77.2	37.7	35.3	69.6	80.3	79.4	98.4	86.4

Appendix 10: Code for training the underlying victim model for the MNIST task

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Step -2: Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Step -1: Define hyperparameters of network and learning algorithm
batch_size = 128
learning_rate = 0.1
momentum = 0.9
dropout_rate = 0.5
num_epochs = 50

# Step 0: Import MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
transform=transforms.ToTensor(), download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
transform=transforms.ToTensor())
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)

# Step 1: Design model and instantiate an instance
class MNISTConvNet(nn.Module):
    def __init__(self):
        super(MNISTConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.conv2 = nn.Conv2d(32, 32, 3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.conv4 = nn.Conv2d(64, 64, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64*4*4, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc3 = nn.Linear(200,10)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = x.view(-1, 64*4*4)
        x = self.dropout(F.relu(self.fc1(x)))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = MNISTConvNet().to(device)

# Step 2: Design loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=momentum)

# Step 3: Train the model
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 200 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}], Loss:
{loss.item():.4f}')

# Step 4: Save the model
print('Finished Training')
PATH = './MNIST_model_trained_yes_dropout.pth'
torch.save(model.state_dict(), PATH)

# Step 5: Test the model
model.eval()
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    n_class_correct = [0 for i in range(10)]
    n_class_samples = [0 for i in range(10)]
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        # max returns (value ,index)
        _, predicted = torch.max(outputs, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()

        for i in range(labels.shape[0]):
            label = labels[i]
            pred = predicted[i]
            if (label == pred):
                n_class_correct[label] += 1
            n_class_samples[label] += 1

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network: {acc} %')

# Step 6: Load the model and test it
load_path = "C:/Users/anon/OneDrive/Documents/Postgraduate_degree/
Individual project/Experiments/MNIST/MNIST_model_trained_yes_dropout.pth"
loaded_model = MNISTConvNet().to(device)
loaded_model.load_state_dict(torch.load(load_path))
loaded_model.eval()
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    n_class_correct = [0 for i in range(10)]
    n_class_samples = [0 for i in range(10)]
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = loaded_model(images)
        # max returns (value ,index)
        _, predicted = torch.max(outputs, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()

        for i in range(labels.shape[0]):
            label = labels[i]
            pred = predicted[i]
            if (label == pred):
                n_class_correct[label] += 1
            n_class_samples[label] += 1

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network: {acc} %')
```

Appendix 11: Code for experimental testing of test-time dropout for the undefended underlying victim model for the MNIST task

```
# Step 1: Mount google drive
from google.colab import drive
drive.mount('/content/drive')

# Step 2: Install foolbox package into environment
!pip install foolbox

# Step 3: Import all required packages
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import foolbox as fb
import numpy as np
import matplotlib.pyplot as plt
import os
import pandas as pd

# Step 4: Add paths to be used in script
data_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST
experiments/data"
pretrained_model_path = "/content/drive/MyDrive/Postgraduate degree individual
project/MNIST experiments/pretrained_models/MNIST_model_trained_yes_dropout.pth"

original_accuracy_path = "/content/drive/MyDrive/Postgraduate degree individual
project/MNIST experiments/Original sample accuracy/undefended_untargeted/"
adv_examples_path = "/content/drive/MyDrive/Postgraduate degree individual
project/MNIST experiments/Adversarial examples/undefended_untargeted/"
adversarial_accuracy_path = "/content/drive/MyDrive/Postgraduate degree individual
project/MNIST experiments/Adversarial examples accuracies
results/undefended_untargeted/"
plots_file_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST
experiments/Adversarial examples accuracies plots/undefended_untargeted/"

path_strings = [original_accuracy_path, adv_examples_path, adversarial_accuracy_path,
plots_file_path]

for path in path_strings:
  if os.path.isdir(path):
    print("{} path already exists".format(path))
  else:
    os.makedirs(path)

# Step 5: Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
# Step 6: Import MNIST test dataset and create sample to be used for generating adversarial
examples
batch_size = 1000
test_dataset = torchvision.datasets.MNIST(root=data_path, train=False,
transform=transforms.ToTensor())
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)
testiter = iter(test_loader)
data_sample = testiter.next()
images = data_sample[0].to(device)
labels = data_sample[1].to(device)

# Step 7: Define dropout rate
dropout_rate = 0

# Step 8: Design model, instantiate an instance, load trained model, and set to undefended
class MNISTConvNet_firstLayer(nn.Module):
    def __init__(self):
        super(MNISTConvNet_firstLayer, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.conv2 = nn.Conv2d(32, 32, 3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.conv4 = nn.Conv2d(64, 64, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 4 * 4, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc3 = nn.Linear(200, 10)
        self.dropout_FC = nn.Dropout(p=0.5)
        self.dropout_CL = nn.Dropout2d(p=dropout_rate)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = x.view(-1, 64 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = MNISTConvNet_firstLayer().to(device)
model.load_state_dict(torch.load(pretrained_model_path))
model.eval()
print(model.training)

# Step 9: Check accuracy of model on original images
# Step 9a: In case of generate:
if len(os.listdir(original_accuracy_path))==0:
  print("if")
  n_samples = labels.size(0)
  undefended_original_outputs = model(images)
  _, undefended_original_predicted = torch.max(undefended_original_outputs, 1)
  undefended_original_n_correct = (undefended_original_predicted == labels).sum().item()
  undefended_original_acc = (undefended_original_n_correct/n_samples)*100.0
  print(undefended_original_acc)
  undefended_original_acc_df = pd.DataFrame(np.array([undefended_original_acc]),
columns=['Original sample accuracy'])
  original_accuracy_path_final = original_accuracy_path + "results"
  undefended_original_acc_df.to_csv(original_accuracy_path_final)

# Step 9b: In case of import:
else:
  print("else")
  original_accuracy_path_final = original_accuracy_path + "results"
  undefended_original_acc_df = pd.read_csv(original_accuracy_path_final, index_col=0)
  print(undefended_original_acc_df)

# Step 10: Generate/import adversarial examples

# Step 10a: Specify epsilons to be used for attacks
epsilons_gradientBased_L2 = np.linspace(0, 10, num=101)
epsilons_gradientBased_Linf = np.linspace(0, 1, num=101)
epsilons_decisionBased_L2 = np.linspace(0, 10, num=101)
epsilons_noiseBased = np.linspace(0, 10, num=101)
epsilons_full_list = [epsilons_gradientBased_L2, epsilons_gradientBased_Linf,
epsilons_decisionBased_L2, epsilons_noiseBased]

# Step 10b: In case of generate:
if len(os.listdir(adv_examples_path))==0:

  print("if")

  # Define gradient-based L2 attacks
  FGSM_L2_attack = fb.attacks.L2FastGradientAttack() # No extra parameters needed
  PGD_L2_attack = fb.attacks.L2ProjectedGradientDescentAttack(rel_stepsize=(1/250),
steps=260) # Requires rel_stepsize and steps as parameters
  DF_L2_attack = fb.attacks.L2DeepFoolAttack(steps=260) # Requires steps, candidates and
overshoot as parameters
  CW_L2_attack = fb.attacks.L2CarliniWagnerAttack(binary_search_steps=20, steps=10000,
stepsize=0.01, initial_const=0.01) # Requies binary_search_steps, steps, stepsize, confidence,
initial_const, abort_early as parameters

  # Define gradient-based Linf attacks
  FGSM_Linf_attack = fb.attacks.LinfFastGradientAttack() # No extra parameters needed
  PGD_Linf_attack = fb.attacks.LinfProjectedGradientDescentAttack(rel_stepsize=(1/250),
steps=260) # Requires rel_stepsize and steps as parameters
  DF_Linf_attack = fb.attacks.LinfDeepFoolAttack(steps=260) # Requires steps, candidates
and overshoot as parameters

  # Define decision-based L2 attack
  Boundary_attack =
fb.attacks.BoundaryAttack(init_attack=fb.attacks.L2CarliniWagnerAttack(), steps=10000) #
Requires step, spherical_step, source_step, source_step_convergance and step_adaptation as
parameters

  # Define noise-based attack
  SandPnoise_attack = fb.attacks.SaltAndPepperNoiseAttack() # Requires steps and across
channels as parameters

  # Create fmodel
  bounds = (0, 1)
  undefended_fmodel = fb.PyTorchModel(model, bounds=bounds)

  # Choose adversary goal
  criterion = fb.criteria.Misclassification(labels)

  # Generate gradient-based L2 adversarial examples for undefended model
  undefended_FGSM_L2_raw, undefended_FGSM_L2_clipped,
undefended_FGSM_L2_is_adv = FGSM_L2_attack(undefended_fmodel, images, criterion,
epsilons=epsilons_gradientBased_L2)
  undefended_PGD_L2_raw, undefended_PGD_L2_clipped, undefended_PGD_L2_is_adv =
PGD_L2_attack(undefended_fmodel, images, criterion,
epsilons=epsilons_gradientBased_L2)
  undefended_DF_L2_raw, undefended_DF_L2_clipped, undefended_DF_L2_is_adv =
DF_L2_attack(undefended_fmodel, images, criterion, epsilons=epsilons_gradientBased_L2)
  undefended_CW_L2_raw, undefended_CW_L2_clipped, undefended_CW_L2_is_adv =
CW_L2_attack(undefended_fmodel, images, criterion, epsilons=epsilons_gradientBased_L2)

  # Generate gradient-based Linf adversarial examples for undefended model
  undefended_FGSM_Linf_raw, undefended_FGSM_Linf_clipped,
undefended_FGSM_Linf_is_adv = FGSM_Linf_attack(undefended_fmodel, images,
criterion, epsilons=epsilons_gradientBased_Linf)
  undefended_PGD_Linf_raw, undefended_PGD_Linf_clipped,
undefended_PGD_Linf_is_adv = PGD_Linf_attack(undefended_fmodel, images, criterion,
epsilons=epsilons_gradientBased_Linf)
  undefended_DF_Linf_raw, undefended_DF_Linf_clipped, undefended_DF_Linf_is_adv =
DF_Linf_attack(undefended_fmodel, images, criterion,
epsilons=epsilons_gradientBased_Linf)

  # Generate decision-based L2 adversarial examples for undefended model
  undefended_Boundary_raw, undefended_Boundary_clipped, undefended_Boundary_is_adv
= Boundary_attack(undefended_fmodel, images, criterion,
epsilons=epsilons_decisionBased_L2)
```

```
  # Generate noise-based adversarial examples for undefended model
  undefended_SandPnoise_raw, undefended_SandPnoise_clipped,
undefended_SandPnoise_is_adv = SandPnoise_attack(undefended_fmodel, images, criterion,
epsilons=epsilons_noiseBased)

  # Save adversarial examples to google drive
  raw_examples_dict = {"FGSM_L2": undefended_FGSM_L2_raw,
                       "PGD_L2": undefended_PGD_L2_raw,
                       "DF_L2": undefended_DF_L2_raw,
                       "CW_L2": undefended_CW_L2_raw,
                       "FGSM_Linf": undefended_FGSM_Linf_raw,
                       "PGD_Linf": undefended_PGD_Linf_raw,
                       "DF_Linf": undefended_DF_Linf_raw,
                       "Boundary": undefended_Boundary_raw,
                       "SandPnoise": undefended_SandPnoise_raw}

  clipped_examples_dict = {"FGSM_L2": undefended_FGSM_L2_clipped,
                           "PGD_L2": undefended_PGD_L2_clipped,
                           "DF_L2": undefended_DF_L2_clipped,
                           "CW_L2": undefended_CW_L2_clipped,
                           "FGSM_Linf": undefended_FGSM_Linf_clipped,
                           "PGD_Linf": undefended_PGD_Linf_clipped,
                           "DF_Linf": undefended_DF_Linf_clipped,
                           "Boundary": undefended_Boundary_clipped,
                           "SandPnoise": undefended_SandPnoise_clipped}

  attacks_full_list = [["FGSM_L2", "PGD_L2", "DF_L2", "CW_L2"], ["FGSM_Linf",
"PGD_Linf", "DF_Linf"], ["Boundary"], ["SandPnoise"]]
  for j, attack_strings in enumerate(attacks_full_list):
    for attack in attack_strings:
      raw_path = adv_examples_path + "raw/" + attack + "/"
      os.makedirs(raw_path)
      clipped_path = adv_examples_path + "clipped/" + attack + "/"
      os.makedirs(clipped_path)
      for i in range(epsilons_full_list[j].shape[0]):
        torch.save(raw_examples_dict[attack][i], (raw_path+str(epsilons_full_list[j][i])))
        torch.save(clipped_examples_dict[attack][i], (clipped_path+str(epsilons_full_list[j][i])))

# Step 10b: In case of importing:
else:

  print("else")

  # Load examples
  raw_examples_dict = {}
  clipped_examples_dict = {}
  attacks_full_list = [["FGSM_L2", "PGD_L2", "DF_L2", "CW_L2"], ["FGSM_Linf",
"PGD_Linf", "DF_Linf"], ["Boundary"], ["SandPnoise"]]
  for j, attack_strings in enumerate(attacks_full_list):
    for attack in attack_strings:
      raw_examples_list = []
      clipped_examples_list = []
      for i in range(epsilons_full_list[j].shape[0]):
        raw_path = adv_examples_path + "raw/" + attack + "/" + str(epsilons_full_list[j][i])
        raw_tensor = torch.load(raw_path)
        raw_examples_list.append(raw_tensor)
        clipped_path = adv_examples_path + "clipped/" + attack + "/" +
str(epsilons_full_list[j][i])
        clipped_tensor = torch.load(clipped_path)
        clipped_examples_list.append(clipped_tensor)
      raw_examples_dict[attack] = raw_examples_list
      clipped_examples_dict[attack] = clipped_examples_list

# # SAVING BOUNDARY ATTACK ADVERSARIAL EXAMPLES
# raw_path = adv_examples_path + "raw/" + "Boundary" + "/"
# os.makedirs(raw_path)
# clipped_path = adv_examples_path + "clipped/" + "Boundary" + "/"
# os.makedirs(clipped_path)
# for i in range(epsilons_decisionBased_L2.shape[0]):
#   torch.save(undefended_Boundary_raw[i],
(raw_path+str(epsilons_decisionBased_L2[i])))
#   torch.save(undefended_Boundary_clipped[i],
(clipped_path+str(epsilons_decisionBased_L2[i])))

# # NEEDS EDITING
# # Step 22: Visualise some original images and adversarial examples for undefended model
# plt.imshow(images[0].cpu().view(28,28), cmap='gray')
# plt.show()
# plt.imshow(undefended_CW_L2_clipped[19][0].cpu().view(28,28), cmap='gray')
# plt.show()
# fig1 = plt.figure()
# for i in range(10):
#   ax = fig1.add_axes([i*0.1, 0, 0.1, 1])
#   ax.imshow(undefended_CW_L2_clipped[19][i].cpu().view(28,28), cmap='gray')
# plt.show()

# Step 11: Check accuracy of model on gradient-based L2 adversarial examples

# Step 11a: In case of generate
if "gradientBased_L2" not in os.listdir(adversarial_accuracy_path):

  print("if")

  n_samples = labels.size(0)
  attacks_full_list_index = 0
  adversarial_gradientBased_L2_acc = np.zeros((epsilons_gradientBased_L2.shape[0],
len(attacks_full_list[attacks_full_list_index])))
  for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      adversarial_outputs = model(epsilon)
      _, adversarial_predicted = torch.max(adversarial_outputs, 1)
      adversarial_predicted = adversarial_predicted.view(1,1000)
      adversarial_n_correct = (adversarial_predicted == labels).sum().item()
      adversarial_acc = (adversarial_n_correct/n_samples)*100
      adversarial_gradientBased_L2_acc[i,j] = adversarial_acc

  adversarial_gradientBased_L2_acc_df = pd.DataFrame(adversarial_gradientBased_L2_acc,
index=epsilons_gradientBased_L2, columns=attacks_full_list[attacks_full_list_index])
  adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_L2"
  adversarial_gradientBased_L2_acc_df.to_csv(adversarial_accuracy_path_final)
  print(adversarial_gradientBased_L2_acc_df)

# Step 11b: In case of import
else:

  print("else")

  adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_L2"
  adversarial_gradientBased_L2_acc_df = pd.read_csv(adversarial_accuracy_path_final,
index_col=0)
  print(adversarial_gradientBased_L2_acc_df)

# Step 12: Check accuracy of model on gradient-based Linf adversarial examples

# Step 12a: In case of generate
if "gradientBased_Linf" not in os.listdir(adversarial_accuracy_path):

  print("if")

  n_samples = labels.size(0)
  attacks_full_list_index = 1
  adversarial_gradientBased_Linf_acc = np.zeros((epsilons_gradientBased_Linf.shape[0],
len(attacks_full_list[attacks_full_list_index])))
  for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      adversarial_outputs = model(epsilon)
      _, adversarial_predicted = torch.max(adversarial_outputs, 1)
      adversarial_predicted = adversarial_predicted.view(1,1000)
      adversarial_n_correct = (adversarial_predicted == labels).sum().item()
      adversarial_acc = (adversarial_n_correct/n_samples)*100
      adversarial_gradientBased_Linf_acc[i,j] = adversarial_acc

  adversarial_gradientBased_Linf_acc_df =
pd.DataFrame(adversarial_gradientBased_Linf_acc, index=epsilons_gradientBased_Linf,
columns=attacks_full_list[attacks_full_list_index])
  adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_Linf"
  adversarial_gradientBased_Linf_acc_df.to_csv(adversarial_accuracy_path_final)
  print(adversarial_gradientBased_Linf_acc_df)

# Step 12b: In case of import
```

```
else:

  print("else")

  adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_Linf"
  adversarial_gradientBased_Linf_acc_df = pd.read_csv(adversarial_accuracy_path_final,
index_col=0)
  print(adversarial_gradientBased_Linf_acc_df)

# Step 13: Check accuracy of model on decision-based L2 adversarial examples

# Step 13a: In case of generate
if "decisionBased_L2" not in os.listdir(adversarial_accuracy_path):

  print("if")

  n_samples = labels.size(0)
  attacks_full_list_index = 2
  adversarial_decisionBased_L2_acc = np.zeros((epsilons_decisionBased_L2.shape[0],
len(attacks_full_list[attacks_full_list_index])))
  for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      adversarial_outputs = model(epsilon)
      _, adversarial_predicted = torch.max(adversarial_outputs, 1)
      adversarial_predicted = adversarial_predicted.view(1,1000)
      adversarial_n_correct = (adversarial_predicted == labels).sum().item()
      adversarial_acc = (adversarial_n_correct/n_samples)*100
      adversarial_decisionBased_L2_acc[i,j] = adversarial_acc

  adversarial_decisionBased_L2_acc_df = pd.DataFrame(adversarial_decisionBased_L2_acc,
index=epsilons_decisionBased_L2, columns=attacks_full_list[attacks_full_list_index])
  adversarial_accuracy_path_final = adversarial_accuracy_path + "decisionBased_L2"
  adversarial_decisionBased_L2_acc_df.to_csv(adversarial_accuracy_path_final)
  print(adversarial_decisionBased_L2_acc_df)

# Step 13b: In case of import
else:

  print("else")

  adversarial_accuracy_path_final = adversarial_accuracy_path + "decisionBased_L2"
  adversarial_decisionBased_L2_acc_df = pd.read_csv(adversarial_accuracy_path_final,
index_col=0)
  print(adversarial_decisionBased_L2_acc_df)

# Step 14: Check accuracy of model on noise-based adversarial examples

# Step 14a: In case of generate:
if "noiseBased" not in os.listdir(adversarial_accuracy_path):
  print("if")

  n_samples = labels.size(0)
  attacks_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK WORKS
  adversarial_noiseBased_acc = np.zeros((epsilons_noiseBased.shape[0],
len(attacks_full_list[attacks_full_list_index])))
  for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      adversarial_outputs = model(epsilon)
      _, adversarial_predicted = torch.max(adversarial_outputs, 1)
      adversarial_predicted = adversarial_predicted.view(1,1000)
      adversarial_n_correct = (adversarial_predicted == labels).sum().item()
      adversarial_acc = (adversarial_n_correct/n_samples)*100
      adversarial_noiseBased_acc[i,j] = adversarial_acc

  adversarial_noiseBased_acc_df = pd.DataFrame(adversarial_noiseBased_acc,
index=epsilons_noiseBased, columns=attacks_full_list[attacks_full_list_index])
  adversarial_accuracy_path_final = adversarial_accuracy_path + "noiseBased"
  adversarial_noiseBased_acc_df.to_csv(adversarial_accuracy_path_final)
  print(adversarial_noiseBased_acc_df)

# Step 14b: In case of import
else:

  print("else")

  adversarial_accuracy_path_final = adversarial_accuracy_path + "noiseBased"
  adversarial_noiseBased_acc_df = pd.read_csv(adversarial_accuracy_path_final,
index_col=0)
  print(adversarial_noiseBased_acc_df)

# Step 15: Plot accuracy of model on gradient-based L2 adversarial examples

if "gradientBased_L2.jpg" not in os.listdir(plots_file_path):

  print("if")

  attacks_full_list_index = 0
  epsilons_full_list_index = 0

  fig = plt.figure()
  ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
  for attack in attacks_full_list[attacks_full_list_index]:
    attack_plot = ax.plot(epsilons_gradientBased_L2,
adversarial_gradientBased_L2_acc_df[attack])
  title_string = "Gradient-based L2 attacks"
  ax.set_title(title_string)
  ax.set_xlabel("Perturbation budget")
  ax.set_ylabel("Model accuracy")
  ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
  ax.set_ylim(bottom=0, top=100)
  legend_strings = ["Fast gradient sign method", "Projected gradient descent", "DeepFool",
"Carlini-Wagner"]
  ax.legend(legend_strings)
  fig.show()
  plots_file_path_final = plots_file_path + "gradientBased_L2.jpg"
  fig.savefig(plots_file_path_final)

else:

  print("else")

  plots_file_path_final = plots_file_path + "gradientBased_L2.jpg"
  image = plt.imread(plots_file_path_final)
  plt.imshow(image)

# Step 16: Plot accuracy of model on gradient-based Linf adversarial examples

if "gradientBased_Linf.jpg" not in os.listdir(plots_file_path):

  print("if")

  attacks_full_list_index = 1
  epsilons_full_list_index = 1

  fig = plt.figure()
  ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
  for attack in attacks_full_list[attacks_full_list_index]:
    attack_plot = ax.plot(epsilons_gradientBased_Linf,
adversarial_gradientBased_Linf_acc_df[attack])
  title_string = "Gradient-based Linf attacks"
  ax.set_title(title_string)
  ax.set_xlabel("Perturbation budget")
  ax.set_ylabel("Model accuracy")
  ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
  ax.set_ylim(bottom=0, top=100)
  legend_strings = ["Fast gradient sign method", "Projected gradient descent", "DeepFool"]
  ax.legend(legend_strings)
  fig.show()
  plots_file_path_final = plots_file_path + "gradientBased_Linf.jpg"
  fig.savefig(plots_file_path_final)

else:

  print("else")

  plots_file_path_final = plots_file_path + "gradientBased_Linf.jpg"
  image = plt.imread(plots_file_path_final)
  plt.imshow(image)

# Step 17: Plot accuracy of model on decision-based L2 adversarial examples

if "decisionBased_L2.jpg" not in os.listdir(plots_file_path):

  print("if")

  attacks_full_list_index = 2
  epsilons_full_list_index = 2

  fig = plt.figure()
  ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
  for attack in attacks_full_list[attacks_full_list_index]:
    attack_plot = ax.plot(epsilons_decisionBased_L2,
adversarial_decisionBased_L2_acc_df[attack])
  title_string = "Decision-based L2 attacks"
  ax.set_title(title_string)
  ax.set_xlabel("Perturbation budget")
  ax.set_ylabel("Model accuracy")
  ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
  ax.set_ylim(bottom=0, top=100)
  legend_strings = ["Boundary attack"]
  ax.legend(legend_strings)
  fig.show()
  plots_file_path_final = plots_file_path + "decisionBased_L2"
  fig.savefig(plots_file_path_final)

else:

  print("else")

  plots_file_path_final = plots_file_path + "decisionBased_L2.jpg"
  image = plt.imread(plots_file_path_final)
  plt.imshow(image)

# Step 18: Plot accuracy of model on noise-based adversarial examples

if "noiseBased.jpg" not in os.listdir(plots_file_path):

  print("if")

  attacks_full_list_index = 3
  epsilons_full_list_index = 3

  fig = plt.figure()
  ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
  for attack in attacks_full_list[attacks_full_list_index]:
    attack_plot = ax.plot(epsilons_noiseBased, adversarial_noiseBased_acc_df[attack])
  title_string = "Noise-based attacks"
  ax.set_title(title_string)
  ax.set_xlabel("Perturbation budget")
  ax.set_ylabel("Model accuracy")
  ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
  ax.set_ylim(bottom=0, top=100)
  legend_strings = ["Salt-and-pepper noise attack"]
  ax.legend(legend_strings)
  fig.show()
  plots_file_path_final = plots_file_path + "noiseBased.jpg"
  fig.savefig(plots_file_path_final)

else:

  print("else")

  plots_file_path_final = plots_file_path + "noiseBased.jpg"
  image = plt.imread(plots_file_path_final)
  plt.imshow(image)
```

Appendix 12: Code template for experimental testing of test-time dropout for a given test-time dropout configuration

Note the following two points:

1. The specific configuration used in this code template is for a dropout rate of 0.1 and for test-time dropout being applied to the 1st layer of the CNN
2. In the supplemental file submission, this code template is provided as a .ipynb file, whereas here it is provided as a .py file. This is due to the inability to format a .ipynb file in a word processing file. The original code template was implemented as a .ipynb file.

```
# Step 1: Mount google drive
from google.colab import drive
drive.mount('/content/drive')

# Step 2: Install foolbox package into environment
!pip install foolbox

# Step 3: Import all required packages
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import foolbox as fb
import numpy as np
import matplotlib.pyplot as plt
import os
import pandas as pd

# Step 4: Add paths to be used in script

data_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST experiments/data"
pretrained_model_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST experiments/pretrained_models/MNIST_model_trained_yes_dropout.pth"

config_string = "1stLayer_0.1/"
original_accuracy_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST experiments/Original sample accuracy/" + config_string
adv_examples_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST experiments/Adversarial examples/" + config_string
adversarial_accuracy_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST experiments/Adversarial examples accuracies results/" + config_string
plots_file_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST experiments/Adversarial examples accuracies plots/" + config_string
improvement_adversarial_accuracy_path = "/content/drive/MyDrive/Postgraduate degree individual project/MNIST experiments/Improvement adversarial examples accuracies results/" + config_string
```

```
improvement_adversarial_plots_path = "/content/drive/MyDrive/Postgraduate degree
individual project/MNIST experiments/Improvement adversarial examples accuracies plots/"
+ config_string

path_strings = [original_accuracy_path, adv_examples_path, adversarial_accuracy_path,
plots_file_path, improvement_adversarial_accuracy_path,
improvement_adversarial_plots_path]

for path in path_strings:
  if os.path.isdir(path):
    print("{} path already exists".format(path))
  else:
    os.makedirs(path)
    print("{} newly created".format(path))

# Step 5: Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)

# Step 6: Import MNIST test dataset and create sample to be used for generating adversarial
examples
batch_size = 1000
test_dataset = torchvision.datasets.MNIST(root=data_path, train=False,
transform=transforms.ToTensor())
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)
testiter = iter(test_loader)
data_sample = testiter.next()
images = data_sample[0].to(device)
labels = data_sample[1].to(device)

# Step 7: Define dropout rate
dropout_rate = 0.1

# Step 8: Design model, instantiate an instance, load trained model, and set to undefended
class MNISTConvNet_firstLayer(nn.Module):
    def __init__(self):
        super(MNISTConvNet_firstLayer, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3)
        self.conv2 = nn.Conv2d(32, 32, 3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.conv4 = nn.Conv2d(64, 64, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 4 * 4, 200)
        self.fc2 = nn.Linear(200, 200)
        self.fc3 = nn.Linear(200, 10)
        self.dropout_FC = nn.Dropout(p=0.5)
        self.dropout_CL = nn.Dropout2d(p=dropout_rate)

    def forward(self, x):
        x = self.dropout_CL(F.relu(self.conv1(x)))
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = x.view(-1, 64 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = MNISTConvNet_firstLayer().to(device)
model.load_state_dict(torch.load(pretrained_model_path))
model.train()
print(model.training)

# Step 9: Generate/import accuracy of model on original images

# Step 9a: In case of generate:
if len(os.listdir(original_accuracy_path))==0:

  print("if")

  n_samples = labels.size(0)

  ensemble_sizes = [3, 7, 15, 21]
  max_ensemble_size = max(ensemble_sizes)
  ensemble_sizes_str = ["3", "7", "15", "21"]
  original_acc_df = pd.DataFrame(np.zeros((1, len(ensemble_sizes_str))),
columns=ensemble_sizes_str)

  all_original_predicted_tensor = torch.zeros(max_ensemble_size, n_samples).to(device)
  for i in range(max_ensemble_size):
    original_outputs = model(images)
    _, original_predicted = torch.max(original_outputs, 1)
    original_predicted = original_predicted.view(1, n_samples)
    all_original_predicted_tensor[i, :] = original_predicted

  for size in ensemble_sizes:
    original_predicted_ensemble = all_original_predicted_tensor[0:size,
:].mode(dim=0).values
    original_n_correct_ensemble = (original_predicted_ensemble == labels).sum().item()
    original_acc_ensemble = (original_n_correct_ensemble/n_samples)*100
    original_acc_df[str(size)] = original_acc_ensemble

  print(original_acc_df)
  original_accuracy_path_final = original_accuracy_path + "results"
  original_acc_df.to_csv(original_accuracy_path_final)
```

```
# Step 9b: In case of import:
else:

  print("else")

  original_accuracy_path_final = original_accuracy_path + "results"
  original_acc_df = pd.read_csv(original_accuracy_path_final, index_col=0)
  print(original_acc_df)

# Step 10: Generate/import adversarial examples

# Step 10a: Specify epsilons to be used for attacks
epsilons_gradientBased_L2 = np.linspace(0, 10, num=101)
epsilons_gradientBased_Linf = np.linspace(0, 1, num=101)
# epsilons_decisionBased_L2 = np.linspace(0, 10, num=101)
epsilons_noiseBased = np.linspace(0, 10, num=101)
epsilons_full_list = [epsilons_gradientBased_L2,
                      epsilons_gradientBased_Linf,
                      # epsilons_decisionBased_L2,
                      epsilons_noiseBased]

# Step 10b: In case of generate:
if len(os.listdir(adv_examples_path))==0:

  print("if")

  # Define gradient-based L2 attacks
  FGSM_L2_attack = fb.attacks.L2FastGradientAttack() # No extra parameters needed
  PGD_L2_attack = fb.attacks.L2ProjectedGradientDescentAttack(rel_stepsize=(1/250),
steps=260) # Requires rel_stepsize and steps as parameters
  DF_L2_attack = fb.attacks.L2DeepFoolAttack(steps=260) # Requires steps, candidates and
overshoot as parameters
  CW_L2_attack = fb.attacks.L2CarliniWagnerAttack(binary_search_steps=20, steps=10000,
stepsize=0.01, initial_const=0.01) # Requies binary_search_steps, steps, stepsize, confidence,
initial_const, abort_early as parameters

  # Define gradient-based Linf attacks
  FGSM_Linf_attack = fb.attacks.LinfFastGradientAttack() # No extra parameters needed
  PGD_Linf_attack = fb.attacks.LinfProjectedGradientDescentAttack(rel_stepsize=(1/250),
steps=260) # Requires rel_stepsize and steps as parameters
  DF_Linf_attack = fb.attacks.LinfDeepFoolAttack(steps=260) # Requires steps, candidates
and overshoot as parameters

  # # Define decision-based L2 attack
  # Boundary_attack = fb.attacks.BoundaryAttack(steps=10000) # Requires step,
spherical_step, source_step, source_step_convergance and step_adaptation as parameters

  # Define noise-based attack
  SandPnoise_attack = fb.attacks.SaltAndPepperNoiseAttack() # Requires steps and across
channels as parameters
```

```
  # Create fmodel
  bounds = (0, 1)
  fmodel = fb.PyTorchModel(model, bounds=bounds)

  # Choose adversary goal
  criterion = fb.criteria.Misclassification(labels)

  # Generate gradient-based L2 adversarial examples
  FGSM_L2_raw, FGSM_L2_clipped, FGSM_L2_is_adv = FGSM_L2_attack(fmodel,
images, criterion, epsilons=epsilons_gradientBased_L2)
  PGD_L2_raw, PGD_L2_clipped, PGD_L2_is_adv = PGD_L2_attack(fmodel, images,
criterion, epsilons=epsilons_gradientBased_L2)
  DF_L2_raw, DF_L2_clipped, DF_L2_is_adv = DF_L2_attack(fmodel, images, criterion,
epsilons=epsilons_gradientBased_L2)
  CW_L2_raw, CW_L2_clipped, CW_L2_is_adv = CW_L2_attack(fmodel, images, criterion,
epsilons=epsilons_gradientBased_L2)

  # Generate gradient-based Linf adversarial examples
  FGSM_Linf_raw, FGSM_Linf_clipped, FGSM_Linf_is_adv = FGSM_Linf_attack(fmodel,
images, criterion, epsilons=epsilons_gradientBased_Linf)
  PGD_Linf_raw, PGD_Linf_clipped, PGD_Linf_is_adv = PGD_Linf_attack(fmodel,
images, criterion, epsilons=epsilons_gradientBased_Linf)
  DF_Linf_raw, DF_Linf_clipped, DF_Linf_is_adv = DF_Linf_attack(fmodel, images,
criterion, epsilons=epsilons_gradientBased_Linf)

  # # Generate decision-based L2 adversarial examples
  # Boundary_raw, Boundary_clipped, Boundary_is_adv = Boundary_attack(fmodel, images,
criterion, epsilons=epsilons_decisionBased_L2)

  # Import transfer-based adversarial examples
  undefended_adversarial_examples_path = "/content/drive/MyDrive/Postgraduate degree
individual project/MNIST experiments/Adversarial
examples/undefended_untargeted/clipped/CW_L2/"
  transfer_clipped = []
  for i in range(epsilons_gradientBased_L2.shape[0]):
    final_path = undefended_adversarial_examples_path + str(epsilons_gradientBased_L2[i])
    final_tensor = torch.load(final_path)
    transfer_clipped.append(final_tensor)

  # Generate noise-based adversarial examples
  SandPnoise_raw, SandPnoise_clipped, SandPnoise_is_adv = SandPnoise_attack(fmodel,
images, criterion, epsilons=epsilons_noiseBased)

  # Save adversarial examples
  raw_examples_dict = {"FGSM_L2": FGSM_L2_raw,
                       "PGD_L2": PGD_L2_raw,
                       "DF_L2": DF_L2_raw,
                       "CW_L2": CW_L2_raw,
                       "FGSM_Linf": FGSM_Linf_raw,
                    "PGD_Linf": PGD_Linf_raw,
                    "DF_Linf": DF_Linf_raw,
                   # "Boundary": Boundary_raw,
                    "SandPnoise": SandPnoise_raw}

  clipped_examples_dict = {"FGSM_L2": FGSM_L2_clipped,
                        "PGD_L2": PGD_L2_clipped,
                        "DF_L2": DF_L2_clipped,
                        "CW_L2": CW_L2_clipped,
                        "FGSM_Linf": FGSM_Linf_clipped,
                        "PGD_Linf": PGD_Linf_clipped,
                        "DF_Linf": DF_Linf_clipped,
                       # "Boundary": Boundary_clipped,
                        "Transfer": transfer_clipped,
                        "SandPnoise": SandPnoise_clipped}

  attacks_full_list = [["FGSM_L2", "PGD_L2", "DF_L2", "CW_L2"],
                    ["FGSM_Linf", "PGD_Linf", "DF_Linf"],
                   # ["Boundary"],
                    ["SandPnoise"]]

  for j, attack_strings in enumerate(attacks_full_list):
    for attack in attack_strings:
      raw_path = adv_examples_path + "raw/" + attack + "/"
      os.makedirs(raw_path)
      clipped_path = adv_examples_path + "clipped/" + attack + "/"
      os.makedirs(clipped_path)
      for i in range(epsilons_full_list[j].shape[0]):
        torch.save(raw_examples_dict[attack][i], (raw_path+str(epsilons_full_list[j][i])))
        torch.save(clipped_examples_dict[attack][i], (clipped_path+str(epsilons_full_list[j][i])))

# Step 10c: In case of importing
else:

  print("else")

  # Load examples
  raw_examples_dict = {}
  clipped_examples_dict = {}
  attacks_full_list = [["FGSM_L2", "PGD_L2", "DF_L2", "CW_L2"],
                    ["FGSM_Linf", "PGD_Linf", "DF_Linf"],
                   # ["Boundary"],
                    ["SandPnoise"]]

  for j, attack_strings in enumerate(attacks_full_list):
    for attack in attack_strings:
      raw_examples_list = []
      clipped_examples_list = []
      for i in range(epsilons_full_list[j].shape[0]):
        raw_path = adv_examples_path + "raw/" + attack + "/" + str(epsilons_full_list[j][i])
        raw_tensor = torch.load(raw_path)
        raw_examples_list.append(raw_tensor)
        clipped_path = adv_examples_path + "clipped/" + attack + "/" +
str(epsilons_full_list[j][i])
        clipped_tensor = torch.load(clipped_path)
        clipped_examples_list.append(clipped_tensor)
      raw_examples_dict[attack] = raw_examples_list
      clipped_examples_dict[attack] = clipped_examples_list

  undefended_adversarial_examples_path = "/content/drive/MyDrive/Postgraduate degree
individual project/MNIST experiments/Adversarial
examples/undefended_untargeted/clipped/CW_L2/"
  transfer_clipped = []
  for i in range(epsilons_gradientBased_L2.shape[0]):
    final_path = undefended_adversarial_examples_path + str(epsilons_gradientBased_L2[i])
    final_tensor = torch.load(final_path)
    transfer_clipped.append(final_tensor)
  clipped_examples_dict["Transfer"] = transfer_clipped

# # NEEDS EDITING
# # Step 22: Visualise some original images and adversarial examples for undefended model
# plt.imshow(images[100].cpu().view(28,28), cmap='gray')
# plt.show()
# plt.imshow(raw_examples_dict["CW_L2"][100][100].cpu().view(28,28), cmap='gray')
# plt.show()
# # fig1 = plt.figure()
# # for i in range(10):
# #   ax = fig1.add_axes([i*0.1, 0, 0.1, 1])
# #   ax.imshow(undefended_CW_L2_clipped[19][i].cpu().view(28,28), cmap='gray')
# # plt.show()

# Step 11: Check accuracy of model on gradient-based L2 adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_L2_" + size for size in ensemble_sizes_str]
directory_files_list = os.listdir(adversarial_accuracy_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 11a: In case of generating:
if not result:

  print("if")

  ensemble_sizes = [3, 7, 15, 21]
  max_ensemble_size = max(ensemble_sizes)
  n_samples = labels.size(0)
  attacks_full_list_index = 0
  epsilons_full_list_index = 0
  base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  adversarial_gradientBased_L2_acc_df_dict = {"3": base_df_3, "7": base_df_7, "15":
base_df_15, "21": base_df_21}

  for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      all_adversarial_predicted_tensor = torch.zeros(max_ensemble_size,
n_samples).to(device)
      for k in range(max_ensemble_size):
        adversarial_outputs = model(epsilon)
        _, adversarial_predicted = torch.max(adversarial_outputs, 1)
        adversarial_predicted = adversarial_predicted.view(1, n_samples)
        all_adversarial_predicted_tensor[k, :] = adversarial_predicted
      for size in ensemble_sizes:
        adversarial_predicted_ensemble = all_adversarial_predicted_tensor[0:size,
:].mode(dim=0).values
        adversarial_n_correct_ensemble = (adversarial_predicted_ensemble ==
labels).sum().item()
        adversarial_acc_ensemble = (adversarial_n_correct_ensemble/n_samples)*100

adversarial_gradientBased_L2_acc_df_dict[str(size)][attack][epsilons_full_list[epsilons_full_
list_index][i]] = adversarial_acc_ensemble

  print(adversarial_gradientBased_L2_acc_df_dict)

  # Code for saving
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_L2_" +
size
    adversarial_gradientBased_L2_acc_df_dict[size].to_csv(adversarial_accuracy_path_final)
```

```
# Step 11b: In case of importing:
else:

  print("else")

  adversarial_gradientBased_L2_acc_df_dict = {}
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_L2_" +
size
    adversarial_gradientBased_L2_acc_df_dict[size] =
pd.read_csv(adversarial_accuracy_path_final, index_col=0)
  print(adversarial_gradientBased_L2_acc_df_dict)

# Step 12: Check accuracy of model on gradient-based Linf adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_Linf_" + size for size in ensemble_sizes_str]
directory_files_list = os.listdir(adversarial_accuracy_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 12a: In case of generating:
if not result:

  print("if")

  ensemble_sizes = [3, 7, 15, 21]
  max_ensemble_size = max(ensemble_sizes)
  n_samples = labels.size(0)
  attacks_full_list_index = 1
  epsilons_full_list_index = 1

  base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
```

```
  base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  adversarial_gradientBased_Linf_acc_df_dict = {"3": base_df_3, "7": base_df_7, "15":
base_df_15, "21": base_df_21}

  for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      all_adversarial_predicted_tensor = torch.zeros(max_ensemble_size,
n_samples).to(device)
      for k in range(max_ensemble_size):
        adversarial_outputs = model(epsilon)
        _, adversarial_predicted = torch.max(adversarial_outputs, 1)
        adversarial_predicted = adversarial_predicted.view(1, n_samples)
        all_adversarial_predicted_tensor[k, :] = adversarial_predicted
      for size in ensemble_sizes:
        adversarial_predicted_ensemble = all_adversarial_predicted_tensor[0:size,
:].mode(dim=0).values
        adversarial_n_correct_ensemble = (adversarial_predicted_ensemble ==
labels).sum().item()
        adversarial_acc_ensemble = (adversarial_n_correct_ensemble/n_samples)*100

adversarial_gradientBased_Linf_acc_df_dict[str(size)][attack][epsilons_full_list[epsilons_ful
l_list_index][i]] = adversarial_acc_ensemble

  print(adversarial_gradientBased_Linf_acc_df_dict)

  # Code for saving
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_Linf_" +
size

adversarial_gradientBased_Linf_acc_df_dict[size].to_csv(adversarial_accuracy_path_final)

# Step 12b: In case of importing:
else:

  print("else")

  adversarial_gradientBased_Linf_acc_df_dict = {}
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "gradientBased_Linf_" +
size
    adversarial_gradientBased_Linf_acc_df_dict[size] =
pd.read_csv(adversarial_accuracy_path_final, index_col=0)
  print(adversarial_gradientBased_Linf_acc_df_dict)

# # Step 13: Check accuracy of model on decision-based L2 adversarial examples

# ensemble_sizes_str = ["3", "7", "15", "21"]
# files_string_list = ["decisionBased_L2_" + size for size in ensemble_sizes_str]
# directory_files_list = os.listdir(adversarial_accuracy_path)
# result = all(elem in directory_files_list for elem in files_string_list)

# # Step 13a: In case of generating:
# if not result:

#   print("if")

#   ensemble_sizes = [3, 7, 15, 21]
#   max_ensemble_size = max(ensemble_sizes)
#   n_samples = labels.size(0)
#   attacks_full_list_index = 2
#   epsilons_full_list_index = 2

#   base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
#   base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
#   base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
#   base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
#   base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   adversarial_decisionBased_L2_acc_df_dict = {"3": base_df_3, "7": base_df_7, "15":
base_df_15, "21": base_df_21}

#   for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
#     for i, epsilon in enumerate(clipped_examples_dict[attack]):
#       all_adversarial_predicted_tensor = torch.zeros(max_ensemble_size,
n_samples).to(device)
#       for k in range(max_ensemble_size):
#         adversarial_outputs = model(epsilon)
#         _, adversarial_predicted = torch.max(adversarial_outputs, 1)
#         adversarial_predicted = adversarial_predicted.view(1,n_samples)
#         all_adversarial_predicted_tensor[k, :] = adversarial_predicted
#       for size in ensemble_sizes:
#         adversarial_predicted_ensemble = all_adversarial_predicted_tensor[0:size,
:].mode(dim=0).values
```

```
#       adversarial_n_correct_ensemble = (adversarial_predicted_ensemble ==
labels).sum().item()
#       adversarial_acc_ensemble = (adversarial_n_correct_ensemble/n_samples)*100
#
adversarial_decisionBased_L2_acc_df_dict[str(size)][attack][epsilons_full_list[epsilons_full
_list_index][i]] = adversarial_acc_ensemble

#   print(adversarial_decisionBased_L2_acc_df_dict)

#   ## Code for saving
#   # for size in ensemble_sizes_str:
#   #   adversarial_accuracy_path_final = adversarial_accuracy_path +
"decisionBased_L2_" + size
#   #
adversarial_decisionBased_L2_acc_df_dict[size].to_csv(adversarial_accuracy_path_final)

# # Step 13b: In case of importing:
# else:

#   print("else")

#   adversarial_decisionBased_L2_acc_df_dict = {}
#   for size in ensemble_sizes_str:
#     adversarial_accuracy_path_final = adversarial_accuracy_path + "decisionBased_L2_"
+ size
#     adversarial_decisionBased_L2_acc_df_dict[size] =
pd.read_csv(adversarial_accuracy_path_final, index_col=0)
#   print(adversarial_decisionBased_L2_acc_df_dict)

# Step 14: Check accuracy of model on transfer-based adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["transferBased_" + size for size in ensemble_sizes_str]
directory_files_list = os.listdir(adversarial_accuracy_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 14a: In case of generating:
if not result:

  print("if")

  ensemble_sizes = [3, 7, 15, 21]
  max_ensemble_size = max(ensemble_sizes)
  n_samples = labels.size(0)

  base_array_3 = np.zeros((epsilons_gradientBased_L2.shape[0], 1))
  base_array_7 = np.zeros((epsilons_gradientBased_L2.shape[0], 1))
  base_array_15 = np.zeros((epsilons_gradientBased_L2.shape[0], 1))
  base_array_21 = np.zeros((epsilons_gradientBased_L2.shape[0], 1))
  base_df_3 = pd.DataFrame(base_array_3, columns=["Transfer"],
index=epsilons_gradientBased_L2)
  base_df_7 = pd.DataFrame(base_array_7, columns=["Transfer"],
index=epsilons_gradientBased_L2)
  base_df_15 = pd.DataFrame(base_array_15, columns=["Transfer"],
index=epsilons_gradientBased_L2)
  base_df_21 = pd.DataFrame(base_array_21, columns=["Transfer"],
index=epsilons_gradientBased_L2)
  adversarial_transferBased_acc_df_dict = {"3": base_df_3, "7": base_df_7, "15":
base_df_15, "21": base_df_21}

  for j, attack in enumerate(["Transfer"]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      all_adversarial_predicted_tensor = torch.zeros(max_ensemble_size,
n_samples).to(device)
      for k in range(max_ensemble_size):
        adversarial_outputs = model(epsilon)
        _, adversarial_predicted = torch.max(adversarial_outputs, 1)
        adversarial_predicted = adversarial_predicted.view(1,n_samples)
        all_adversarial_predicted_tensor[k, :] = adversarial_predicted
      for size in ensemble_sizes:
        adversarial_predicted_ensemble = all_adversarial_predicted_tensor[0:size,
:].mode(dim=0).values
        adversarial_n_correct_ensemble = (adversarial_predicted_ensemble ==
labels).sum().item()
        adversarial_acc_ensemble = (adversarial_n_correct_ensemble/n_samples)*100
        adversarial_transferBased_acc_df_dict[str(size)][attack][epsilons_gradientBased_L2[i]]
= adversarial_acc_ensemble

  print(adversarial_transferBased_acc_df_dict)

  # Code for saving
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "transferBased_" + size
    adversarial_transferBased_acc_df_dict[size].to_csv(adversarial_accuracy_path_final)

# Step 14b: In case of importing:
else:

  print("else")

  adversarial_transferBased_acc_df_dict = {}
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "transferBased_" + size
    adversarial_transferBased_acc_df_dict[size] =
pd.read_csv(adversarial_accuracy_path_final, index_col=0)
  print(adversarial_transferBased_acc_df_dict)

# Step 15: Check accuracy of model on noise-based adversarial examples
```

```
ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["noiseBased_" + size for size in ensemble_sizes_str]
directory_files_list = os.listdir(adversarial_accuracy_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 15a: In case of generating:
if not result:

  print("if")

  ensemble_sizes = [3, 7, 15, 21]
  max_ensemble_size = max(ensemble_sizes)
  n_samples = labels.size(0)
  attacks_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK WORKS
  epsilons_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK WORKS

  base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  adversarial_noiseBased_acc_df_dict = {"3": base_df_3, "7": base_df_7, "15": base_df_15,
"21": base_df_21}

  for j, attack in enumerate(attacks_full_list[attacks_full_list_index]):
    for i, epsilon in enumerate(clipped_examples_dict[attack]):
      all_adversarial_predicted_tensor = torch.zeros(max_ensemble_size,
n_samples).to(device)
      for k in range(max_ensemble_size):
        adversarial_outputs = model(epsilon)
        _, adversarial_predicted = torch.max(adversarial_outputs, 1)
        adversarial_predicted = adversarial_predicted.view(1,n_samples)
        all_adversarial_predicted_tensor[k, :] = adversarial_predicted
      for size in ensemble_sizes:
        adversarial_predicted_ensemble = all_adversarial_predicted_tensor[0:size,
:].mode(dim=0).values
        adversarial_n_correct_ensemble = (adversarial_predicted_ensemble ==
labels).sum().item()
        adversarial_acc_ensemble = (adversarial_n_correct_ensemble/n_samples)*100

adversarial_noiseBased_acc_df_dict[str(size)][attack][epsilons_full_list[epsilons_full_list_in
dex][i]] = adversarial_acc_ensemble

  print(adversarial_noiseBased_acc_df_dict)

  # Code for saving
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "noiseBased_" + size
    adversarial_noiseBased_acc_df_dict[size].to_csv(adversarial_accuracy_path_final)

# Step 15b: In case of importing:
else:

  print("else")

  adversarial_noiseBased_acc_df_dict = {}
  for size in ensemble_sizes_str:
    adversarial_accuracy_path_final = adversarial_accuracy_path + "noiseBased_" + size
    adversarial_noiseBased_acc_df_dict[size] = pd.read_csv(adversarial_accuracy_path_final,
index_col=0)
  print(adversarial_noiseBased_acc_df_dict)

# Step 16: Plot accuracy of model on gradient-based L2 adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_L2_" + size + ".jpg" for size in ensemble_sizes_str]
directory_files_list = os.listdir(plots_file_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 16a: In case of generating
if not result:

  print("if")

  attacks_full_list_index = 0
  epsilons_full_list_index = 0
  legend_strings = ["Fast gradient sign method", "Projected gradient descent", "DeepFool",
"Carlini-Wagner"]

  for size in ensemble_sizes_str:
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    for attack in attacks_full_list[attacks_full_list_index]:
      attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
adversarial_gradientBased_L2_acc_df_dict[size][attack])
      title_string = "Gradient-based L2 attacks ensemble size " + size
      ax.set_title(title_string)
      ax.set_xlabel("Perturbation budget")
      ax.set_ylabel("Model accuracy")
      ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
      ax.set_ylim(bottom=0, top=100)
      ax.legend(legend_strings)
    fig.show()
    plots_file_path_final = plots_file_path + "gradientBased_L2_" + size + ".jpg"
    fig.savefig(plots_file_path_final)

# Step 16b: In case of importing
else:

  print("else")

  for size in ensemble_sizes_str:
    plots_file_path_final = plots_file_path + "gradientBased_L2_" + size + ".jpg"
    image = plt.imread(plots_file_path_final)
    plt.imshow(image)

# Step 17: Plot accuracy of model on gradient-based Linf adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_Linf_" + size + ".jpg" for size in ensemble_sizes_str]
directory_files_list = os.listdir(plots_file_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 17a: In case of generating
if not result:

  print("if")

  attacks_full_list_index = 1
  epsilons_full_list_index = 1
  legend_strings = ["Fast gradient sign method", "Projected gradient descent", "DeepFool"]

  for size in ensemble_sizes_str:
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    for attack in attacks_full_list[attacks_full_list_index]:
      attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
adversarial_gradientBased_Linf_acc_df_dict[size][attack])
      title_string = "Gradient-based Linf attacks ensemble size " + size
      ax.set_title(title_string)
      ax.set_xlabel("Perturbation budget")
      ax.set_ylabel("Model accuracy")
      ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
      ax.set_ylim(bottom=0, top=100)
      ax.legend(legend_strings)
    fig.show()
    plots_file_path_final = plots_file_path + "gradientBased_Linf_" + size + ".jpg"
    fig.savefig(plots_file_path_final)

# Step 17b: In case of importing
else:

  print("else")

  for size in ensemble_sizes_str:
    plots_file_path_final = plots_file_path + "gradientBased_Linf_" + size + ".jpg"
    image = plt.imread(plots_file_path_final)
    plt.imshow(image)

# # Step 18: Plot accuracy of model on decision-based L2 adversarial examples

# ensemble_sizes_str = ["3", "7", "15", "21"]
# files_string_list = ["decisionBased_L2_" + size + ".jpg" for size in ensemble_sizes_str]
# directory_files_list = os.listdir(plots_file_path)
# result = all(elem in directory_files_list for elem in files_string_list)

# # Step 18a: In case of generating
# if not result:

#   print("if")

#   attacks_full_list_index = 2
#   epsilons_full_list_index = 2
#   legend_strings = ["Boundary attack"]

#   for size in ensemble_sizes_str:
#     fig = plt.figure()
#     ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
#     for attack in attacks_full_list[attacks_full_list_index]:
#       attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
adversarial_decisionBased_L2_acc_df_dict[size][attack])
#       title_string = "Decision-based L2 attacks ensemble size " + size
#       ax.set_title(title_string)
#       ax.set_xlabel("Perturbation budget")
#       ax.set_ylabel("Model accuracy")
#       ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
#       ax.set_ylim(bottom=0, top=100)
#       ax.legend(legend_strings)
#     fig.show()
#     plots_file_path_final = plots_file_path + "decisionBased_L2_" + size + ".jpg"
#     # fig.savefig(plots_file_path_final)

# # Step 18b: In case of importing
# else:

#   print("else")

#   for size in ensemble_sizes_str:
#     plots_file_path_final = plots_file_path + "decisionBased_L2_" + size + ".jpg"
#     image = plt.imread(plots_file_path_final)
#     plt.imshow(image)

# Step 19: Plot accuracy of model on transfer-based adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["transferBased_" + size + ".jpg" for size in ensemble_sizes_str]
directory_files_list = os.listdir(plots_file_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 19a: In case of generating
if not result:

  print("if")

  legend_strings = ["Transfer-based attack"]

  for size in ensemble_sizes_str:
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    for attack in ["Transfer"]:
      attack_plot = ax.plot(epsilons_gradientBased_L2,
adversarial_transferBased_acc_df_dict[size][attack])
      title_string = "Transfer-based attacks ensemble size " + size
      ax.set_title(title_string)
      ax.set_xlabel("Perturbation budget")
      ax.set_ylabel("Model accuracy")
      ax.set_xlim(left=epsilons_gradientBased_L2[0] , right=epsilons_gradientBased_L2[-1])
      ax.set_ylim(bottom=0, top=100)
      ax.legend(legend_strings)
    fig.show()
    plots_file_path_final = plots_file_path + "transferBased_" + size + ".jpg"
    fig.savefig(plots_file_path_final)

# Step 19b: In case of importing
else:

  print("else")

  for size in ensemble_sizes_str:
    plots_file_path_final = plots_file_path + "transferBased_" + size + ".jpg"
    image = plt.imread(plots_file_path_final)
    plt.imshow(image)

# Step 20: Plot accuracy of model on noise-based adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["noiseBased_" + size + ".jpg" for size in ensemble_sizes_str]
directory_files_list = os.listdir(plots_file_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 20a: In case of generating
if not result:

  print("if")

  attacks_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK STARTS
WORKING
  epsilons_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK STARTS
WORKING
  legend_strings = ["Salt-and-pepper noise attack"]

  for size in ensemble_sizes_str:
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    for attack in attacks_full_list[attacks_full_list_index]:
      attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
adversarial_noiseBased_acc_df_dict[size][attack])
      title_string = "Noise-based attacks ensemble size " + size
      ax.set_title(title_string)
      ax.set_xlabel("Perturbation budget")
      ax.set_ylabel("Model accuracy")
      ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
      ax.set_ylim(bottom=0, top=100)
      ax.legend(legend_strings)
    fig.show()
    plots_file_path_final = plots_file_path + "noiseBased_" + size + ".jpg"
    fig.savefig(plots_file_path_final)

# Step 20b: In case of importing
else:

  print("else")

  for size in ensemble_sizes_str:
    plots_file_path_final = plots_file_path + "noiseBased_" + size + ".jpg"
    image = plt.imread(plots_file_path_final)
    plt.imshow(image)
```

```
# Step 21: Determine improvement over undefended model for gradient-based L2 adversarial
attacks

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_L2_" + size for size in ensemble_sizes_str]
directory_files_list = os.listdir(improvement_adversarial_accuracy_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 21a: In case of generating
if not result:

  print("if")

  # import undefended model
  undefended_adversarial_acc_path = "/content/drive/MyDrive/Postgraduate degree
individual project/MNIST experiments/Adversarial examples accuracies
results/undefended_untargeted/"
  undefended_adversarial_acc_path_final = undefended_adversarial_acc_path +
"gradientBased_L2"
  undefended_adversarial_gradientBased_L2_acc_df =
pd.read_csv(undefended_adversarial_acc_path_final, index_col=0)
  undefended_adversarial_gradientBased_L2_acc_df.index =
adversarial_gradientBased_L2_acc_df_dict["3"].index

  # Measure improvement in accuracy
  attacks_full_list_index = 0
  epsilons_full_list_index = 0

  base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  improvement_adversarial_gradientBased_L2_acc_df_dict = {"3": base_df_3, "7":
base_df_7, "15": base_df_15, "21": base_df_21}
```

```
  for size in ensemble_sizes_str:
    for attack in attacks_full_list[attacks_full_list_index]:
      improvement = adversarial_gradientBased_L2_acc_df_dict[size][attack] -
undefended_adversarial_gradientBased_L2_acc_df[attack] # MIGHT NEED EDITING TO
ACCOUNT FOR DIFFERING INDEXES WHEN IMPORTING UNDEFENDED
ACCURACIES
      improvement_adversarial_gradientBased_L2_acc_df_dict[size][attack] = improvement

  print(improvement_adversarial_gradientBased_L2_acc_df_dict)

  # Code for saving
  for size in ensemble_sizes_str:
    improvement_adversarial_accuracy_path_final = improvement_adversarial_accuracy_path
+ "gradientBased_L2_" + size

improvement_adversarial_gradientBased_L2_acc_df_dict[size].to_csv(improvement_adversa
rial_accuracy_path_final)

# Step 21b: In case of importing
else:

  print="else"

  improvement_adversarial_gradientBased_L2_acc_df_dict = {}
  for size in ensemble_sizes_str:
    improvement_adversarial_accuracy_path_final = improvement_adversarial_accuracy_path
+ "gradientBased_L2_" + size
    improvement_adversarial_gradientBased_L2_acc_df_dict[size] =
pd.read_csv(improvement_adversarial_accuracy_path_final)
  print(improvement_adversarial_gradientBased_L2_acc_df_dict)

# Step 22: Determine improvement over undefended model for gradient-based Linf
adversarial attacks

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_Linf_" + size for size in ensemble_sizes_str]
directory_files_list = os.listdir(improvement_adversarial_accuracy_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 22a: In case of generating
if not result:

  print("if")

  # import undefended model
  undefended_adversarial_acc_path = "/content/drive/MyDrive/Postgraduate degree
individual project/MNIST experiments/Adversarial examples accuracies
results/undefended_untargeted/"
```

```
  undefended_adversarial_acc_path_final = undefended_adversarial_acc_path +
"gradientBased_Linf"
  undefended_adversarial_gradientBased_Linf_acc_df =
pd.read_csv(undefended_adversarial_acc_path_final, index_col=0)
  undefended_adversarial_gradientBased_Linf_acc_df.index =
adversarial_gradientBased_Linf_acc_df_dict["3"].index

  # Measure improvement in accuracy
  attacks_full_list_index = 1
  epsilons_full_list_index = 1

  base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  improvement_adversarial_gradientBased_Linf_acc_df_dict = {"3": base_df_3, "7":
base_df_7, "15": base_df_15, "21": base_df_21}

  for size in ensemble_sizes_str:
    for attack in attacks_full_list[attacks_full_list_index]:
      improvement = adversarial_gradientBased_Linf_acc_df_dict[size][attack] -
undefended_adversarial_gradientBased_Linf_acc_df[attack] # MIGHT NEED EDITING TO
ACCOUNT FOR DIFFERING INDEXES WHEN IMPORTING UNDEFENDED
ACCURACIES
      improvement_adversarial_gradientBased_Linf_acc_df_dict[size][attack] = improvement

  print(improvement_adversarial_gradientBased_Linf_acc_df_dict)

  # Code for saving
  for size in ensemble_sizes_str:
    improvement_adversarial_accuracy_path_final = improvement_adversarial_accuracy_path
+ "gradientBased_Linf_" + size
```

```
improvement_adversarial_gradientBased_Linf_acc_df_dict[size].to_csv(improvement_adver
sarial_accuracy_path_final)

# Step 22b: In case of importing
else:

  print="else"

  ensemble_sizes_str = ["1", "3", "5", "7"]
  improvement_adversarial_gradientBased_Linf_acc_df_dict = {}
  for size in ensemble_sizes_str:
    improvement_adversarial_accuracy_path_final = improvement_adversarial_accuracy_path
+ "gradientBased_Linf_" + size
    improvement_adversarial_gradientBased_Linf_acc_df_dict[size] =
pd.read_csv(improvement_adversarial_accuracy_path_final)

  print(improvement_adversarial_gradientBased_Linf_acc_df_dict)

# # Step 23: Determine improvement over undefended model for decision-based L2
adversarial attacks

# ensemble_sizes_str = ["3", "7", "15", "21"]
# files_string_list = ["decisionBased_L2_" + size for size in ensemble_sizes_str]
# directory_files_list = os.listdir(improvement_adversarial_accuracy_path)
# result = all(elem in directory_files_list for elem in files_string_list)

# # Step 23a: In case of generating
# if not result:

#   print("if")

#   # import undefended model
#   undefended_adversarial_acc_path = "/content/drive/MyDrive/Postgraduate degree
individual project/MNIST experiments/Adversarial examples accuracies
results/undefended_untargeted/"
#   undefended_adversarial_acc_path_final = undefended_adversarial_acc_path +
"decisionBased_L2"
#   undefended_adversarial_decisionBased_L2_acc_df =
pd.read_csv(undefended_adversarial_acc_path_final, index_col=0)
#   undefended_adversarial_decisionBased_L2_acc_df.index =
adversarial_decisionBased_L2_acc_df_dict["3"].index

#   # Measure improvement in accuracy
#   attacks_full_list_index = 2
#   epsilons_full_list_index = 2

#   base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
```

```
#   base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
#   base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
#   base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
#   base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
#   improvement_adversarial_decisionBased_L2_acc_df_dict = {"3": base_df_3, "7":
base_df_7, "15": base_df_15, "21": base_df_21}

#   for size in ensemble_sizes_str:
#     for attack in attacks_full_list[attacks_full_list_index]:
#       improvement = adversarial_decisionBased_L2_acc_df_dict[size][attack] -
undefended_adversarial_decisionBased_L2_acc_df[attack] # MIGHT NEED EDITING TO
ACCOUNT FOR DIFFERING INDEXES WHEN IMPORTING UNDEFENDED
ACCURACIES
#       improvement_adversarial_decisionBased_L2_acc_df_dict[size][attack] = improvement

#   print(improvement_adversarial_decisionBased_L2_acc_df_dict)

#   # # Code for saving
#   # for size in ensemble_sizes_str:
#   #   improvement_adversarial_accuracy_path_final =
improvement_adversarial_accuracy_path + "decisionBased_L2_" + size
#   #
improvement_adversarial_decisionBased_L2_acc_df_dict[size].to_csv(improvement_advers
arial_accuracy_path_final)

# # Step 23b: In case of importing
# else:

#   print="else"

#   improvement_adversarial_decisionBased_L2_acc_df_dict = {}
#   for size in ensemble_sizes_str:
#     improvement_adversarial_accuracy_path_final =
improvement_adversarial_accuracy_path + "decisionBased_L2_" + size
#     improvement_adversarial_decisionBased_L2_acc_df_dict[size] =
pd.read_csv(improvement_adversarial_accuracy_path_final)
```

```
#   print(improvement_adversarial_decisionBased_L2_acc_df_dict)

# Step 24: Determine improvement over undefended model for noise-based L2 adversarial
attacks

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["noiseBased_" + size for size in ensemble_sizes_str]
directory_files_list = os.listdir(improvement_adversarial_accuracy_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 24a: In case of generating
if not result:

  print("if")

  # import undefended model
  undefended_adversarial_acc_path = "/content/drive/MyDrive/Postgraduate degree
individual project/MNIST experiments/Adversarial examples accuracies
results/undefended_untargeted/"
  undefended_adversarial_acc_path_final = undefended_adversarial_acc_path + "noiseBased"
  undefended_adversarial_noiseBased_acc_df =
pd.read_csv(undefended_adversarial_acc_path_final, index_col=0)
  undefended_adversarial_noiseBased_acc_df.index =
adversarial_noiseBased_acc_df_dict["3"].index

  # Measure improvement in accuracy
  attacks_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK WORKS
  epsilons_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK WORKS

  base_array_3 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_7 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_15 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_array_21 = np.zeros((epsilons_full_list[epsilons_full_list_index].shape[0],
len(attacks_full_list[attacks_full_list_index])))
  base_df_3 = pd.DataFrame(base_array_3,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_7 = pd.DataFrame(base_array_7,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_15 = pd.DataFrame(base_array_15,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
  base_df_21 = pd.DataFrame(base_array_21,
columns=attacks_full_list[attacks_full_list_index],
index=epsilons_full_list[epsilons_full_list_index])
```

```
  improvement_adversarial_noiseBased_acc_df_dict = {"3": base_df_3, "7": base_df_7, "15": 
base_df_15, "21": base_df_21}

  for size in ensemble_sizes_str:
    for attack in attacks_full_list[attacks_full_list_index]:
      improvement = adversarial_noiseBased_acc_df_dict[size][attack] - 
undefended_adversarial_noiseBased_acc_df[attack] # MIGHT NEED EDITING TO 
ACCOUNT FOR DIFFERING INDEXES WHEN IMPORTING UNDEFENDED 
ACCURACIES
      improvement_adversarial_noiseBased_acc_df_dict[size][attack] = improvement

  print(improvement_adversarial_noiseBased_acc_df_dict)

  # Code for saving
  for size in ensemble_sizes_str:
    improvement_adversarial_accuracy_path_final = improvement_adversarial_accuracy_path 
+ "noiseBased_" + size

improvement_adversarial_noiseBased_acc_df_dict[size].to_csv(improvement_adversarial_ac
curacy_path_final)

# Step 24b: In case of importing
else:

  print="else"

  improvement_adversarial_noiseBased_acc_df_dict = {}
  for size in ensemble_sizes_str:
    improvement_adversarial_accuracy_path_final = improvement_adversarial_accuracy_path 
+ "noiseBased_" + size
    improvement_adversarial_noiseBased_acc_df_dict[size] = 
pd.read_csv(improvement_adversarial_accuracy_path_final)

  print(improvement_adversarial_noiseBased_acc_df_dict)

# Step 25: Plot accuracy improvement from defence for gradient-based L2 adversarial 
examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_L2_" + size + ".jpg" for size in ensemble_sizes_str]
directory_files_list = os.listdir(improvement_adversarial_plots_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 25a: In case of generating
if not result:

  print("if")

  attacks_full_list_index = 0
  epsilons_full_list_index = 0
  legend_strings = ["Fast gradient sign method", "Projected gradient descent", "DeepFool",
"Carlini-Wagner"]

  for size in ensemble_sizes_str:
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    title_string = "Gradient-based L2 attacks ensemble size " + size
    ax.set_title(title_string)
    ax.set_xlabel("Perturbation budget")
    ax.set_ylabel("Accuracy improvement")
    ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
    # ax.set_ylim(bottom=0, top=100)
    for attack in attacks_full_list[attacks_full_list_index]:
      attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
improvement_adversarial_gradientBased_L2_acc_df_dict[size][attack])
      ax.legend(legend_strings)
    fig.show()
    improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"gradientBased_L2_" + size + ".jpg"
    fig.savefig(improvement_adversarial_plots_path_final)

# Step 25b: In case of importing
else:

  print("else")

  for size in ensemble_sizes_str:
    improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"gradientBased_L2_" + size + ".jpg"
    image = plt.imread(improvement_adversarial_plots_path_final)
    plt.imshow(image)

# Step 26: Plot accuracy improvement from defence for gradient-based Linf adversarial
examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["gradientBased_Linf_" + size + ".jpg" for size in ensemble_sizes_str]
directory_files_list = os.listdir(improvement_adversarial_plots_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 26a: In case of generating
if not result:

  print("if")

  attacks_full_list_index = 1
  epsilons_full_list_index = 1
  legend_strings = ["Fast gradient sign method", "Projected gradient descent", "DeepFool"]
  for size in ensemble_sizes_str:
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    title_string = "Gradient-based Linf attacks ensemble size " + size
    ax.set_title(title_string)
    ax.set_xlabel("Perturbation budget")
    ax.set_ylabel("Accuracy improvement")
    ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
    # ax.set_ylim(bottom=0, top=100)
    for attack in attacks_full_list[attacks_full_list_index]:
      attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
improvement_adversarial_gradientBased_Linf_acc_df_dict[size][attack])
      ax.legend(legend_strings)
    fig.show()
    improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"gradientBased_Linf_" + size + ".jpg"
    fig.savefig(improvement_adversarial_plots_path_final)

# Step 26b: In case of importing
else:

  print("else")

  for size in ensemble_sizes_str:
    improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"gradientBased_Linf_" + size + ".jpg"
    image = plt.imread(improvement_adversarial_plots_path_final)
    plt.imshow(image)

# # Step 27: Plot accuracy improvement from defence for decision-based L2 adversarial
examples

# ensemble_sizes_str = ["3", "7", "15", "21"]
# files_string_list = ["decisionBased_L2_" + size + ".jpg" for size in ensemble_sizes_str]
# directory_files_list = os.listdir(improvement_adversarial_plots_path)
# result = all(elem in directory_files_list for elem in files_string_list)

# # Step 27a: In case of generating
# if not result:

#   print("if")

#   attacks_full_list_index = 2
#   epsilons_full_list_index = 2
#   legend_strings = ["Boundary"]

#   for size in ensemble_sizes_str:
#     fig = plt.figure()
#     ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
#     title_string = "Decision-based L2 attacks ensemble size " + size
#     ax.set_title(title_string)
#     ax.set_xlabel("Perturbation budget")
#     ax.set_ylabel("Accuracy improvement")
#     ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
#     # ax.set_ylim(bottom=0, top=100)
#     for attack in attacks_full_list[attacks_full_list_index]:
#       attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
improvement_adversarial_decisionBased_L2_acc_df_dict[size][attack])
#       ax.legend(legend_strings)
#     fig.show()
#     improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"decisionBased_L2_" + size + ".jpg"
#     # fig.savefig(improvement_adversarial_plots_path_final)

# # Step 27b: In case of importing
# else:

#   print("else")

#   for size in ensemble_sizes_str:
#     improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"decisionBased_L2_" + size + ".jpg"
#     image = plt.imread(improvement_adversarial_plots_path_final)
#     plt.imshow(image)

# Step 28: Plot accuracy improvement from defence for noise-based adversarial examples

ensemble_sizes_str = ["3", "7", "15", "21"]
files_string_list = ["noiseBased_" + size + ".jpg" for size in ensemble_sizes_str]
directory_files_list = os.listdir(improvement_adversarial_plots_path)
result = all(elem in directory_files_list for elem in files_string_list)

# Step 28a: In case of generating
if not result:

  print("if")

  attacks_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK WORKS
  epsilons_full_list_index = 2 # SHOULD BE 3 WHEN BOUNDARY ATTACK WORKS
  legend_strings = ["Salt-and-pepper attack"]

  for size in ensemble_sizes_str:
    fig = plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    title_string = "Noise-based attacks ensemble size " + size
    ax.set_title(title_string)
    ax.set_xlabel("Perturbation budget")
    ax.set_ylabel("Accuracy improvement")
    ax.set_xlim(left=epsilons_full_list[epsilons_full_list_index][0] ,
right=epsilons_full_list[epsilons_full_list_index][-1])
    # ax.set_ylim(bottom=0, top=100)
    for attack in attacks_full_list[attacks_full_list_index]:
      attack_plot = ax.plot(epsilons_full_list[epsilons_full_list_index],
improvement_adversarial_noiseBased_acc_df_dict[size][attack])
      ax.legend(legend_strings)
    fig.show()
    improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"noiseBased_" + size + ".jpg"
    fig.savefig(improvement_adversarial_plots_path_final)

# Step 28b: In case of importing
else:

  print("else")

  for size in ensemble_sizes_str:
    improvement_adversarial_plots_path_final = improvement_adversarial_plots_path +
"noiseBased_" + size + ".jpg"
    image = plt.imread(improvement_adversarial_plots_path_final)
    plt.imshow(image)
```