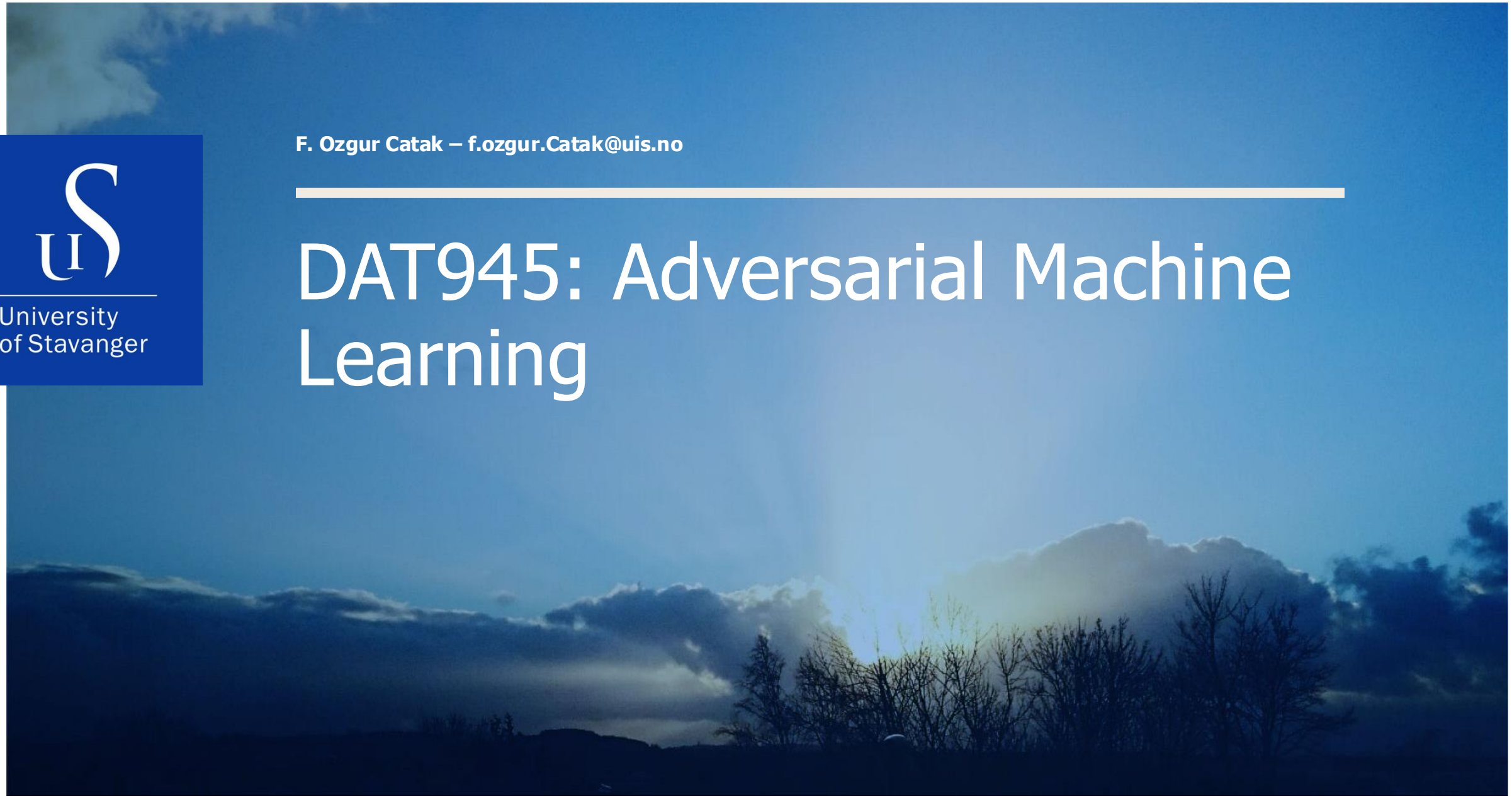


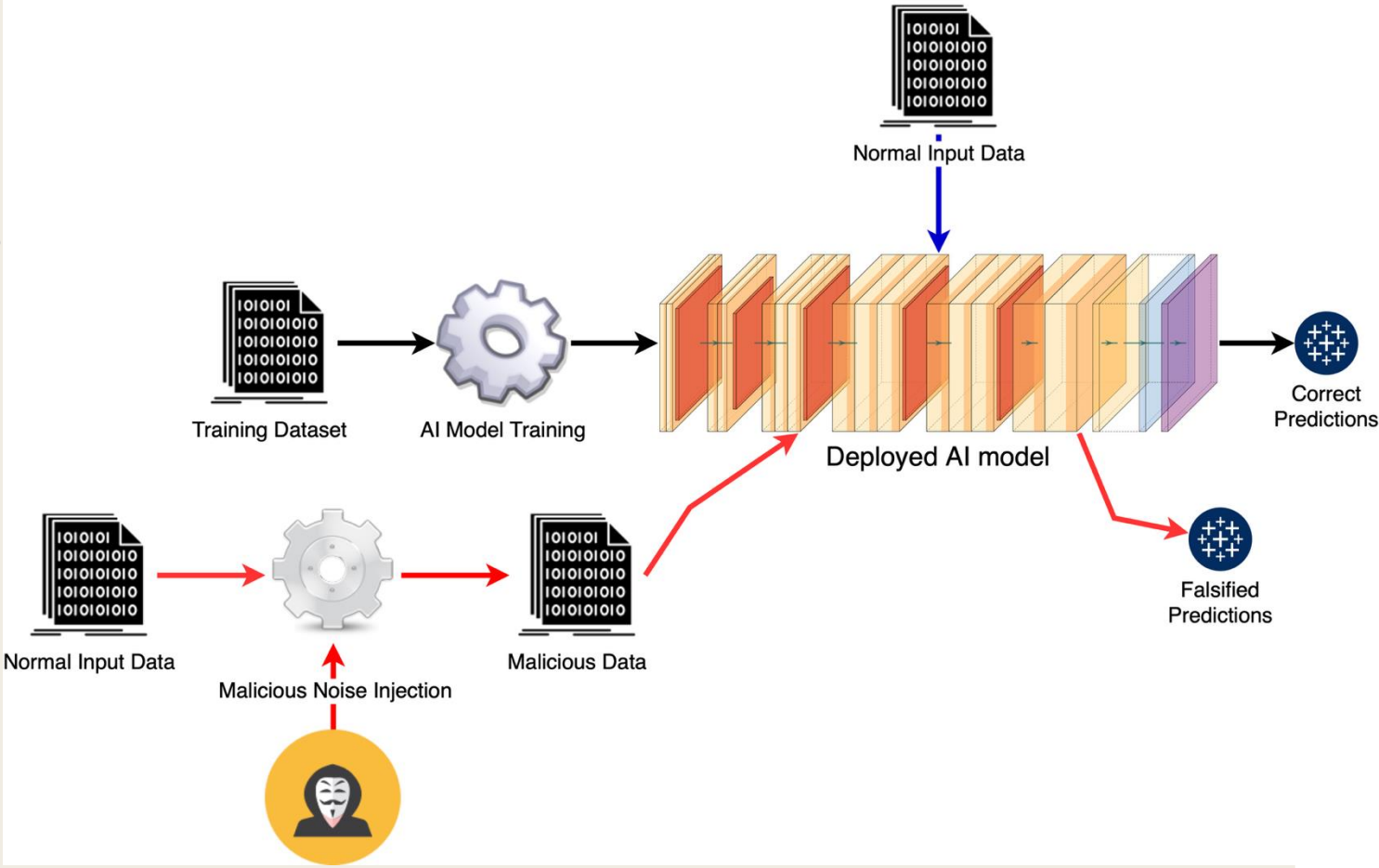
University of Stavanger

F. Ozgur Catak – f.ozgur.Catak@uis.no

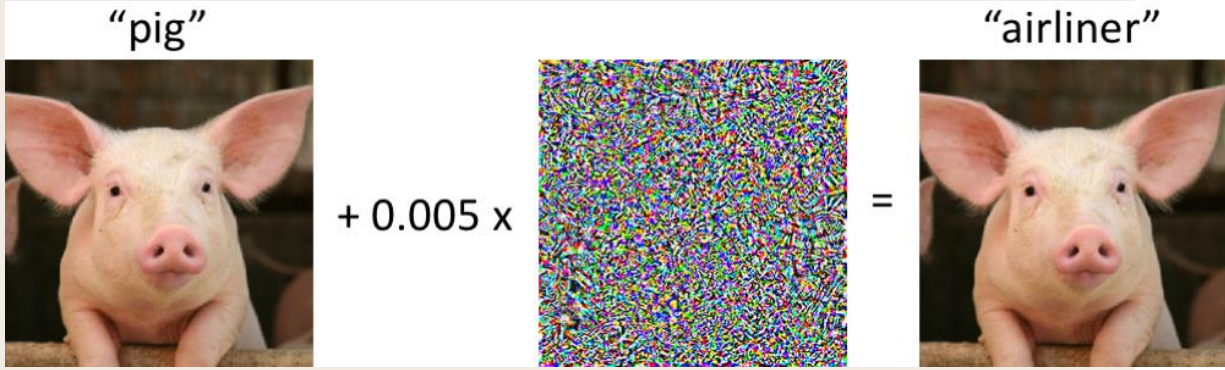
DAT945: Adversarial Machine Learning

Adversarial ML

- Someone tries to fool AI model by making small, clever changes to what it sees.
- Like giving it an optical illusion - what looks like a stop sign to us might look like something else to the computer because of these tricky changes.
- Researchers are constantly working on ways to make them more robust so they can avoid falling for these tricks.
- Solution: giving extra lessons to recognize regular signs and signs with manipulated stickers.



Adversarial Machine Learning: Hacking AI Models



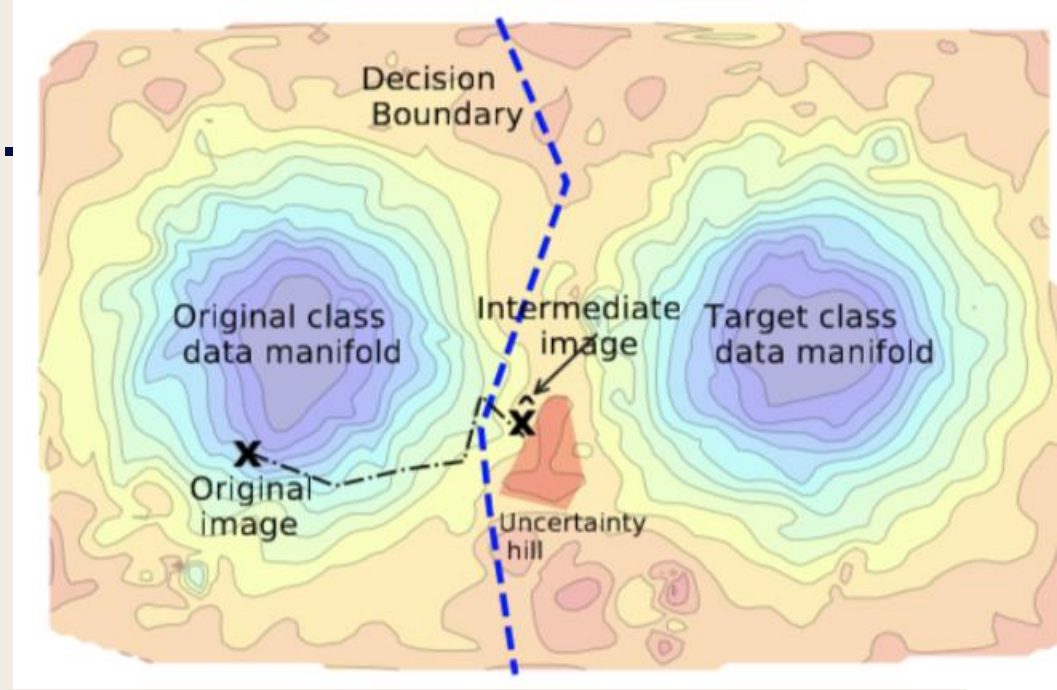
- **Adv. ML is not GAN. They are completely different.**
- DNN models are highly vulnerable for the craftly designed malicious inputs.
- DNN models contain many vulnerabilities and weaknesses which make them difficult to defend in the context of adversarial machine learning.
- For instance, they are often sensitive to small changes in the input data, resulting in unexpected results in the model's final output.
- how an adversary would exploit such a vulnerability and manipulate the model through the use of carefully crafted perturbation applied to the input data.

Adv ML Attacks

- Fast-Gradient Sign Method
- Iterative Gradient Sign Method
- Projected Gradient Descent
- Jacobian-Based Saliency Map
- Carlini & Wagner

Preliminaries: Decision Boundaries

- The key concept of adversarial examples is to fool a DNN model's internal decision boundaries.
- **Training**: DNN learns decision boundaries, which determine its behavior.
  - represented by the network's parameters.
  - **Loss Surface**
  - If an attacker finds out where the decision boundaries are, they can change the input examples in such a way that "pushes" them over the boundary.



White-Box Model: FGSM Attack

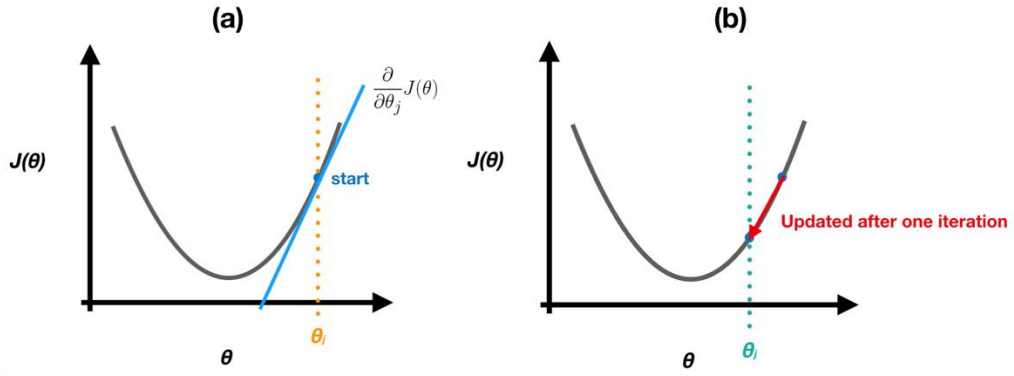
Generating adversarial examples with the Fast Gradient Sign Method [1]

Traditional DNN learning (Gradient Descent)

Repeat until converge {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

} where j represents the feature index number.



$$\theta = \theta - n * \nabla_\theta \mathcal{L}(\theta, x, y)$$

Attacker's aim to manipulate

- DNN parameters should stay the same
- Attack can change the input itself, **x**
- the goal is now to increase the model error.
- the difference to the true class should be large.

$$x' = x + \epsilon * sign(\nabla_x \mathcal{L}(\theta, x, y))$$

```
def create_adversarial_pattern(input_image, input_label):
  with tf.GradientTape() as tape:
    tape.watch(input_image)
    prediction = pretrained_model(input_image)
    loss = loss_object(input_label, prediction)
  # Get the gradients of the loss w.r.t to the input image.
  gradient = tape.gradient(loss, input_image)
  # Get the sign of the gradients to create the perturbation
  signed_grad = tf.sign(gradient)
  return signed_grad
```

Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." *arXiv preprint arXiv:1412.6572* (2014).

Adversarial Machine Learning

- A subfield of ML that deals with the **robustness** of ML algorithms against adversarial attacks.
- Designed to mislead ML models by **manipulating input data** in a way that is imperceptible to humans.
- These attacks can cause ML models to **make incorrect predictions** with potentially harmful consequences.

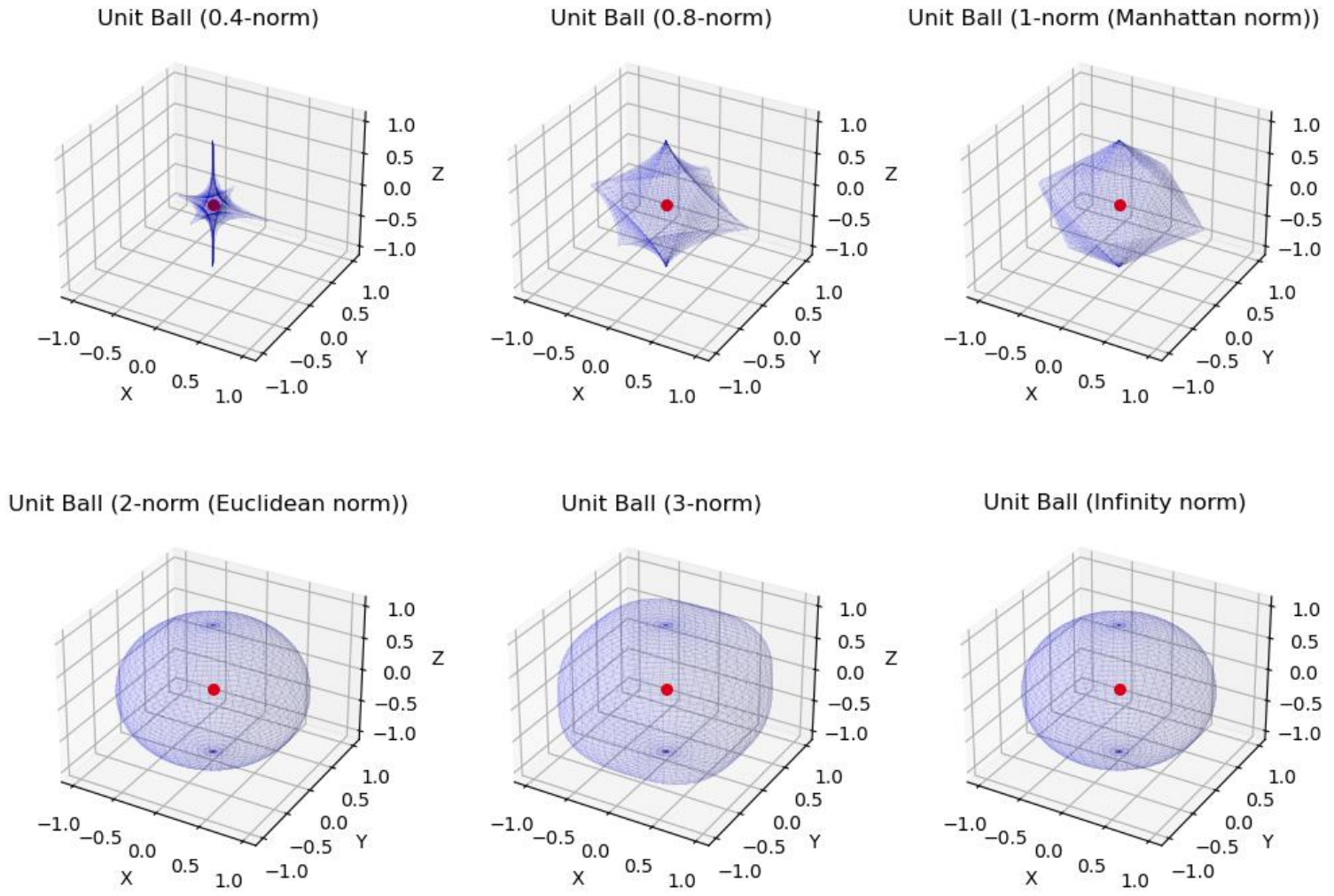
Lp Balls

- Denoted as $Bp(r)$, is a set of points in a vector space that are bounded by a certain distance metric called the **Lp norm**.
- The Lp norm measures the magnitude of a vector

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$$

- The Lp ball is defined as the set of all points x in the vector space
- Such that the Lp norm of x is less than or equal to a given radius r.

$$B_p(r) = \{x \in \mathbb{R}^n : \|\mathbf{x}\|_p \leq r\}$$



Lp Balls

The shape of the Lp ball depends on the value of p. Let's consider a few special cases:

1. **L1 Ball (Manhattan Ball):**
   - The $L1$ ball, denoted as $B_1(r)$, consists of all points in the vector space whose $L1$ norm is less than or equal to r.
   - Mathematically, $B_1(r)$ is defined as:

$$B_1(r) = \{x \in \mathbb{R}^n : \|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \leq r\}$$

   - The $L1$ ball is a polytope with flat edges and corners. In two dimensions, it has the shape of a diamond, and in higher dimensions, it forms a hyperdiamond.
2. **L2 Ball (Euclidean Ball):**
   - The L2 ball, denoted as B2(r), consists of all points in the vector space whose L2 norm is less than or equal to r.
   - B2(r) is defined as:

$$B_2(r) = x \in \mathbb{R}^n : |\mathbf{x}|2 = \sqrt{\sum i = 1^n |x_i|^2} \leq r$$

   - The L2 ball is a sphere in Euclidean space. In two dimensions, it is a circle, and in three dimensions, it is a regular sphere. In higher dimensions, it forms a hypersphere.
   - The L2 norm is also known as the Euclidean norm and represents the length or magnitude of a vector. It is derived from the Pythagorean theorem.
3. $L\infty$ **Ball (Chebyshev Ball):**
   - The $L\infty$ ball, denoted as $B\infty(r)$, consists of all points in the vector space whose $L\infty$ norm is less than or equal to r.
   - $B\infty(r)$ is defined as:

$$B_\infty(r) = x \in \mathbb{R}^n : |\mathbf{x}|\infty = \max 1 \leq i \leq n|x_i| \leq r$$

   - The $L\infty$ ball is a hyperrectangle or hypercube. It is defined by the maximum absolute value of any component of the vector. In two dimensions, it is a square, and in three dimensions, it is a cube. In higher dimensions, it forms a hypercube.
   - The $L\infty$norm represents the maximum absolute value of the elements in a vector. It is also known as the Chebyshev norm or the supremum norm.

Lp Balls and Adv. ML

- The concept of Lp balls is closely related to adversarial ML
- It provides a **geometric interpretation of the boundaries** within which perturbations can be applied to the input data while remaining imperceptible to humans.
- By leveraging the Lp norms, researchers have defined different types of adversarial attacks that aim to find **perturbations that maximize the effect on the model's output** while staying within these prescribed bounds.
- The L2 and L-infty balls are commonly used to define the allowable perturbation regions for crafting adversarial examples.
- **Clipping values**

Cleverhans

Cleverhans is a Python library for adversarial machine learning. It provides tools and functionalities that aid in the research and implementation of adversarial attacks and defenses.

Here are some key features and components of the Cleverhans library:

1. **Adversarial Attack Algorithms:** Cleverhans offers various attack algorithms, including the Fast Gradient Sign Method (FGSM), Projected Gradient Descent (PGD), Basic Iterative Method (BIM), Carlini and Wagner attack (CW), and more. These algorithms can be used to generate adversarial examples for both targeted and untargeted attacks.
2. **Model Wrappers:** Cleverhans provides wrappers for popular deep-learning frameworks like TensorFlow and PyTorch. These wrappers allow users to easily apply adversarial attacks to their models and evaluate their robustness against such attacks.
3. **Adversarial Training:** The library offers functionality for adversarial training, which involves training models using adversarial examples to improve their robustness against attacks. This can help enhance the model's ability to withstand adversarial perturbations and improve overall security.
4. **Evaluation Metrics:** Cleverhans includes metrics for evaluating the robustness of machine learning models against adversarial attacks. These metrics provide insights into the model's performance and vulnerability, helping researchers and practitioners assess the effectiveness of defense mechanisms.
5. **Preprocessing and Defense Mechanisms:** The library offers utilities for data preprocessing and implementing various defense mechanisms against adversarial attacks. These mechanisms include defensive distillation, feature squeezing, spatial smoothing, and more.
6. **Adversarial Examples Datasets:** Cleverhans provides access to adversarial example datasets that can be used for evaluation and benchmarking purposes. These datasets contain adversarial examples generated using different attack algorithms and can help test the robustness of models.

Cleverhans simplifies the process of researching, implementing, and evaluating adversarial attacks and defenses. It provides a wide range of functionalities, making it a valuable tool for studying and understanding the vulnerabilities of machine learning models and developing strategies to enhance their security. The research community and practitioners have widely used the library to explore and address the challenges posed by adversarial attacks in machine learning.

https://github.com/cleverhans-lab/cleverhans

Untargeted Attacks

2.1 Untargeted Attacks

Untargeted attacks aim to deceive the model by causing misclassification of the input data, without a specific target class in mind. The goal of an untargeted attack is to find an adversarial example that maximizes the model's prediction error while not necessarily aligning with any particular class label. In an untargeted attack, the objective can be represented as:

$$\underset{x+\delta}{\text{argmax}} \ \text{Loss}(f(x+\delta), y)$$

where $f(\cdot)$ represents the machine learning model, x is the original input, δ is the adversarial perturbation, y is the true label, and $\text{Loss}(\cdot)$ is a suitable loss function. The attacker's goal is to find the perturbation δ that maximizes the loss and leads to misclassification of the input.

Untargeted attacks are particularly concerning as they can compromise the integrity and reliability of machine learning systems. For instance, an autonomous vehicle misclassifying a stop sign as a speed limit sign due to an untargeted attack could lead to dangerous consequences.

In untargeted attacks, the attacker aims to find a perturbation δ that can be added to the original input x to create a new input $x' = x + \delta$, such that the machine learning model f misclassifies the perturbed input x'. The objective of the attacker is to maximize the loss function $L(f(x'), y)$, where y is the true label of the original input x.

We can represent the problem of untargeted attacks as an optimization problem, which can be formulated as follows:

$$\delta^* = \text{argmax}_\delta \ L(f(x+\delta), y)$$
$$\text{subject to } ||\delta||_p \leq \epsilon$$

where δ^* is the optimal perturbation, ϵ is a pre-defined perturbation budget, and $|| \cdot ||p$ is a norm function used to measure the magnitude of the perturbation. The norm function can be L_1, L_2 or $L\infty$ norm, depending on the application.

Untargeted Attacks - Examples

1. **Adversarial Examples for Image Classification:** In image classification tasks, an attacker can craft adversarial examples that appear visually similar to the original images but cause misclassification by the model. For example, an attacker can apply FGSM or PGD to modify the pixel values of an image slightly, leading to a misclassification by the model. The perturbed image might look almost identical to the original image to the human eye, but the model makes an incorrect prediction.
2. **Speech Recognition Attacks:** Adversarial attacks can also be applied to speech recognition systems. By introducing imperceptible perturbations to the audio waveform, an attacker can cause the speech recognition model to transcribe the audio incorrectly. These attacks can be formulated as untargeted attacks, where the goal is to maximize the loss and induce misclassification.
3. **Text-based Attacks:** Adversarial attacks can target natural language processing (NLP) models as well. By making subtle modifications to the input text, an attacker can cause the model to misclassify or produce incorrect outputs. For example, in sentiment analysis, an attacker can modify a positive sentence to make it misclassified as negative. FGSM, PGD, or other optimization-based methods can be used to generate the adversarial text examples.
4. **Object Detection Attacks:** Object detection systems can also be vulnerable to untargeted attacks. By perturbing the input image, an attacker can cause the model to detect or fail to detect certain objects in the scene. These attacks can have serious implications in applications such as autonomous driving, where misclassifying objects can lead to accidents or safety hazards.
5. **Malware and Intrusion Detection:** Adversarial attacks can be launched against malware and intrusion detection systems as well. By crafting malicious software or network traffic with carefully designed perturbations, an attacker can evade detection or cause false alarms, compromising the security of the system.

Comparison of Attack Methods

Method	Effectiveness	Speed	Complexity	Iterative	Notes
FGSM	Medium	Fast	Low	No	Single-step gradient-based attack
BIM	High	Slow	Medium	Yes	Iterative FGSM
MIM	High	Slow	Medium	Yes	BIM with momentum
Carlini-Wagner	Very High	Slow	High	Yes	Optimization-based, highly targeted

Fast Gradient Sign Method

The Fast Gradient Sign Method (FGSM) is a popular technique for crafting adversarial examples in the field of adversarial machine learning. FGSM is a type of untargeted attack where an attacker adds a small perturbation to the input data to fool a machine learning model.

Formally, let x be an input sample to a machine learning model with parameters θ, and y be the corresponding true label of x. The objective of FGSM is to craft an adversarial example x' such that the model predicts a different label y' than the true label y with the constraint that the perturbation is bounded by a small value ϵ.

The perturbation added to x to create the adversarial example x' is defined as:

$$\eta = \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

where $\nabla_x J(\theta, x, y)$ is the gradient of the loss function $J(\theta, x, y)$ with respect to x. The $sign$ function computes the sign of each element in the gradient, and the resulting perturbation is scaled by the value of ϵ.

The adversarial example is then created by adding the perturbation to the original input:

$$x' = x + \eta$$

where x' is the adversarial example and x is the original input.

Intuitively, the FGSM attack works by using the gradient information of the model to find the direction in which the loss function can be maximally increased for a given input sample. By adding a small perturbation in this direction, the model is tricked into predicting a wrong label.

FGSM is a fast and computationally efficient method for generating adversarial examples, which has made it a popular choice for attacking machine learning models in both research and real-world settings. However, it has some limitations, such as being susceptible to gradient masking, where the model's gradient can be intentionally or unintentionally manipulated to prevent the attacker from creating effective adversarial examples.

FGSM Code Example

- Jupyter Notebook

```
# Generate adversarial examples using FGSM
epsilon = 0.1
adv_x_test = fast_gradient_method(model, x_test, epsilon, np.inf)

# Get predicted labels for the test set
y_pred = model.predict(x_test, verbose=0).argmax(axis=1)
adv_y_pred = model.predict(adv_x_test, verbose=0).argmax(axis=1)
```

Epsilon in Adversarial Attacks

- Epsilon (ε) controls the magnitude of the perturbation applied to the input data.
- **Role of Epsilon**
  - **Magnitude Control**: determines how much noise is added to the original input. A small epsilon value results in a slight perturbation, whereas a large epsilon can lead to more noticeable and significant changes.
  - **Balance**: It balances between making the perturbation imperceptible to humans and ensuring it is sufficient to fool the machine learning model.

Other Names and Interpretations

- Perturbation Budget
- Attack Strength
- Perturbation Magnitude

Practical Considerations

$$x_{adv}^{(i+1)} = \text{Clip}_{x,\epsilon}\{x_{adv}^{(i)} + \alpha \cdot \text{sign}(\nabla_x J(\theta, x_{adv}^{(i)}, y))\}$$

- **Choosing Epsilon**:
  - **Small**: Make the perturbations nearly invisible, but might not fool a well-trained model.
  - **Large**: Can easily fool the model but may produce noticeable changes, making the adversarial nature detectable.
- **Effectiveness vs. Detectability**:
  - Higher can increase the attack's success rate but may make the adv example perceptible to humans or detectable by adversarial defenses.
- **Domain Dependence**:
  - In image-based attacks, epsilon values are often in the range of 0 to 1 (assuming pixel values are normalized).
  - For text or other modalities, the interpretation and scaling of epsilon might differ.

Improved FGSM: Basic Iterative Method (BIM)

3.2 Basic Iterative Method

The BIM attack is an iterative variant of the FGSM attack that aims to generate stronger adversarial examples by taking multiple small steps in the direction of the gradient. It follows a similar approach as FGSM, but instead of making a single update to the input, it performs multiple iterations to refine the adversarial perturbation.

Here's a step-by-step breakdown of the BIM attack using math notation:

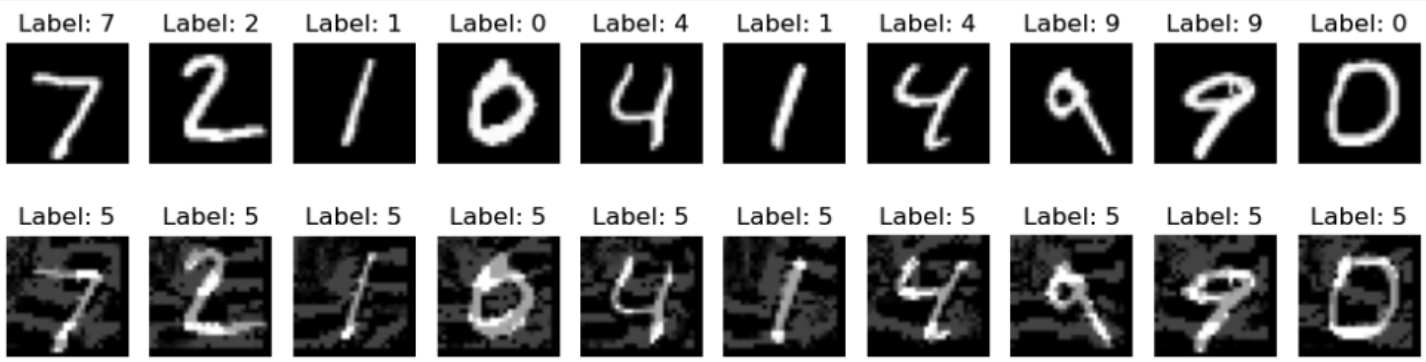
1. Given an input image x, its true label y_{true}, a targeted class y_{target} (optional), and a maximum perturbation size ϵ:
2. Initialize the adversarial example x_{adv} as a copy of the original input x.
3. Repeat for a fixed number of iterations T:
   A. Compute the gradient of the loss function $J(\theta, x_{\text{adv}}, y_{\text{true}})$ with respect to the input: $\nabla_{x_{\text{adv}}} J(\theta, x_{\text{adv}}, y_{\text{true}})$
   B. Compute the perturbation by taking a small step in the direction of the gradient: $\delta = \epsilon \cdot \text{sign}(\nabla_{x_{\text{adv}}} J(\theta, x_{\text{adv}}, y_{\text{true}}))$
   C. Update the adversarial example by adding the perturbation: $x_{\text{adv}} = x_{\text{adv}} + \delta$
   D. Clip the adversarial example to ensure it stays within the ϵ-ball around the original input: $x_{\text{adv}} = \text{clip}(x_{\text{adv}}, x - \epsilon, x + \epsilon)$
4. Return the final adversarial example x_{adv}.

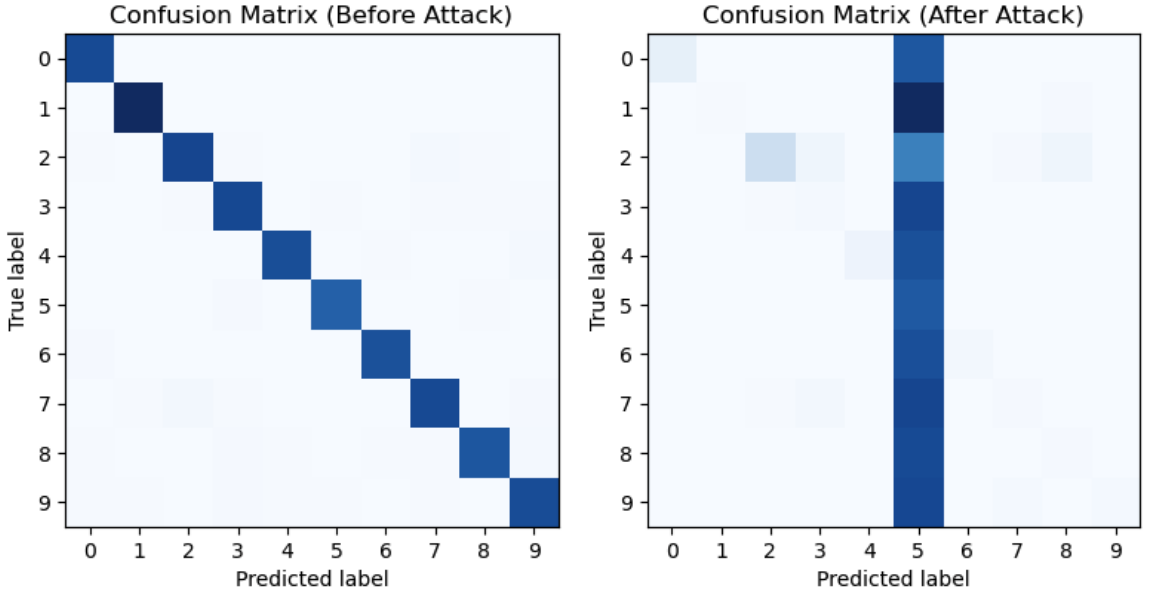
Now, let's discuss the main differences between FGSM and BIM:

1. **Iteration:** FGSM performs a single update to the input using the sign of the gradient, while BIM iteratively updates the adversarial example using multiple small steps in the direction of the gradient.
2. **Strength:** BIM generates stronger adversarial examples compared to FGSM by refining the perturbation over multiple iterations. This allows BIM to explore the input space more effectively and find perturbations that can lead to misclassification.
3. **Defense detection:** Due to the iterative nature, BIM is more likely to be detected by certain defense mechanisms that can detect repeated patterns or multiple modifications to the input. FGSM, on the other hand, may produce less noticeable perturbations and can be less susceptible to detection.
4. **Computational cost:** BIM requires more computational resources compared to FGSM due to its iterative nature. Each iteration involves computing the gradient and updating the adversarial example, which can increase the overall runtime.

BIM Code Example

```
adv_x_test = basic_iterative_method(model, x_test, eps=epsilon, eps_iter=0.01, nb_iter=100,
                                    clip_min=0.0, clip_max=1.0, norm=np.inf, y=target_adv,
                                    targeted=True, sanity_checks=False)
```





Further Improvement: MIM

MIM incorporates the concept of momentum, inspired by optimization algorithms used in deep learning, to improve the efficiency and effectiveness of the attack. Here's a step-by-step breakdown of the MIM attack using math notation:

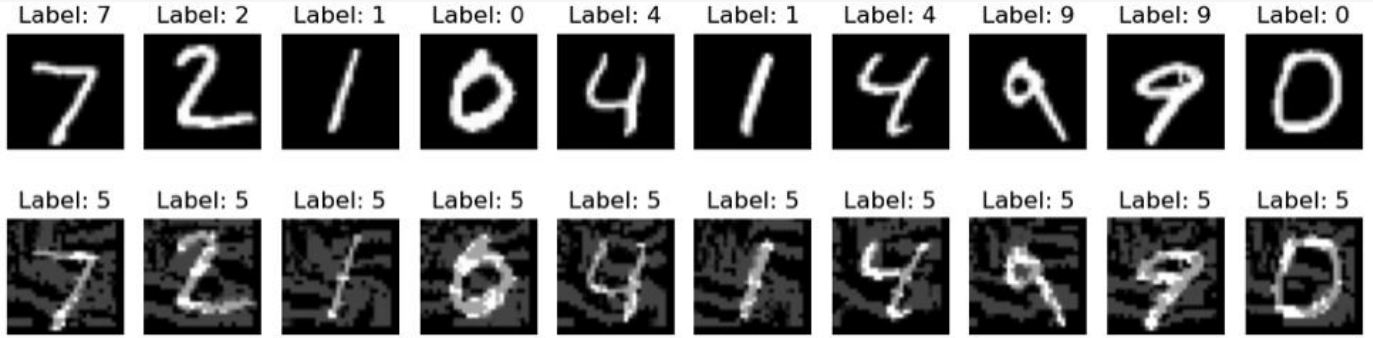
1. Given an input image x, its true label y_{true}, a targeted class y_{target} (optional), a maximum perturbation size ϵ, a momentum factor μ, and a decay factor α:
2. Initialize the adversarial example x_{adv} as a copy of the original input x.
3. Initialize the momentum term v as zero, representing the accumulated gradient direction.
4. Repeat for a fixed number of iterations T:
   A. Compute the gradient of the loss function $J(\theta, x_{\text{adv}}, y_{\text{true}})$ with respect to the input: $\nabla_{x_{\text{adv}}} J(\theta, x_{\text{adv}}, y_{\text{true}})$
   B. Update the momentum term: $v = \mu \cdot v + \frac{\nabla_{x_{\text{adv}}} J(\theta, x_{\text{adv}}, y_{\text{true}})}{|\nabla_{x_{\text{adv}}} J(\theta, x_{\text{adv}}, y_{\text{true}})|_1}$
   C. Compute the perturbation by scaling the momentum term: $\delta = \epsilon \cdot \text{sign}(v)$
   D. Update the adversarial example by adding the perturbation: $x_{\text{adv}} = x_{\text{adv}} + \delta$
   E. Clip the adversarial example to ensure it stays within the ϵ-ball around the original input: $x_{\text{adv}} = \text{clip}(x_{\text{adv}}, x - \epsilon, x + \epsilon)$
   F. Decay the momentum term to gradually reduce the influence of past gradients: $v = \alpha \cdot v$
5. Return the final adversarial example x_{adv}.

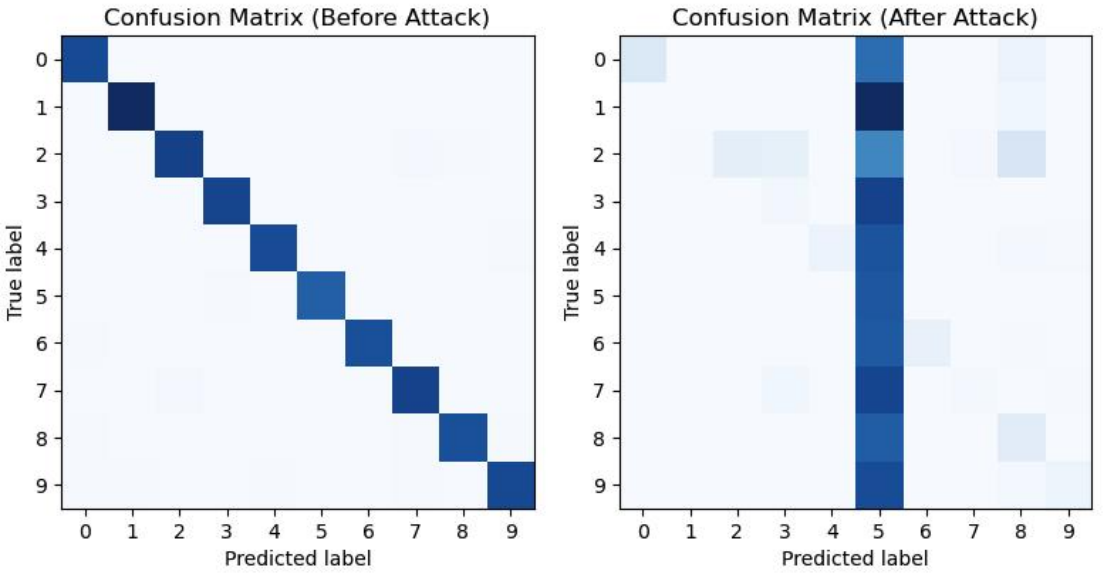
Now, let's discuss the main differences between FGSM and MIM:

1. **Momentum:** MIM introduces a momentum term that accumulates the gradient information over iterations. This allows the attack to maintain a sense of direction and exploit more information from the gradient landscape, potentially leading to more effective perturbations.
2. **Iterative refinement:** Similar to BIM, MIM performs multiple iterations to refine the adversarial perturbation. This iterative refinement enables MIM to explore the input space more comprehensively and potentially find stronger perturbations.
3. **Efficiency:** The use of momentum in MIM can accelerate the convergence of the attack, reducing the number of iterations required to achieve a similar level of perturbation as FGSM. This can lead to improved efficiency compared to both FGSM and BIM.
4. **Defense detection:** Like BIM, the iterative nature of MIM can make it more susceptible to detection by certain defense mechanisms that can identify repeated patterns or multiple modifications to the input. However, MIM's momentum can introduce additional challenges for some detection methods, making it potentially more resilient in certain scenarios.
5. **Computational cost:** MIM requires additional computations for maintaining and updating the momentum term compared to FGSM. While it is more efficient than BIM, the computational cost of MIM is still higher than that of FGSM due to the iterative nature of the attack.

MIM Code Example

```
adv_x_test = momentum_iterative_method(model, x_test, eps=epsilon, eps_iter=0.01, nb_iter=100,
                                       decay_factor=1.0, clip_min=0.0, clip_max=1.0, y=target_adv,
                                       targeted=True, sanity_checks=False)
```





Carlini & Wagner – Optimization Based Attack

The Carlini-Wagner (CW) Attack is a state-of-the-art optimization-based adversarial attack that aims to find the smallest possible perturbation to the input image that will cause the model to misclassify it. The CW attack is unique in that it is capable of producing adversarial examples that are specifically designed to target a certain misclassification, as opposed to simply finding any misclassification.

The CW attack formulates the problem of finding an adversarial example as an optimization problem. Specifically, given an input image x, the attack aims to find a perturbation δ that minimizes the distance between the perturbed image $x + \delta$ and the original image x subject to the constraint that the perturbed image is misclassified by the model. The optimization problem can be formulated as:

$$\min_{\delta} \|\delta\|_p + c \cdot f(x + \delta)$$

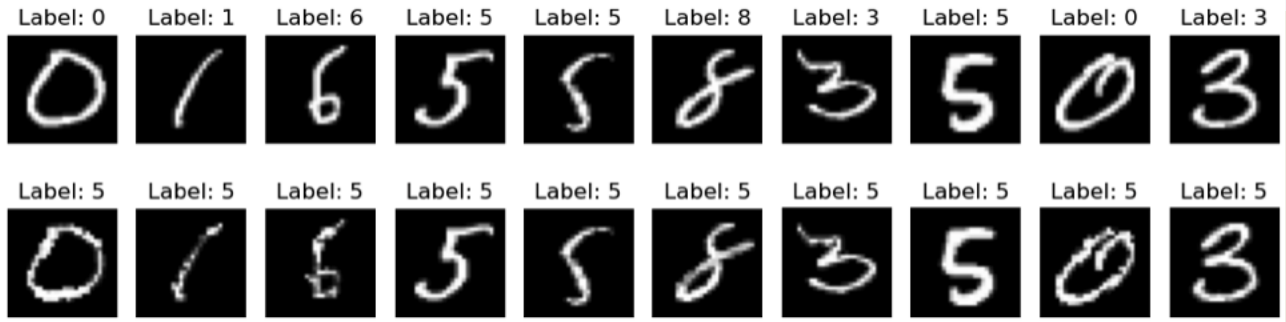
where $\| \cdot \|_p$ is a norm used to measure the size of the perturbation, $f(x + \delta)$ is a function that measures the model's confidence in its prediction for the perturbed image, and c is a trade-off parameter that controls the importance of the two terms in the objective function. The $f(x + \delta)$ term in the objective function is designed to encourage the perturbed image to be classified as the target class, or to be misclassified if targeted attacks are not used.

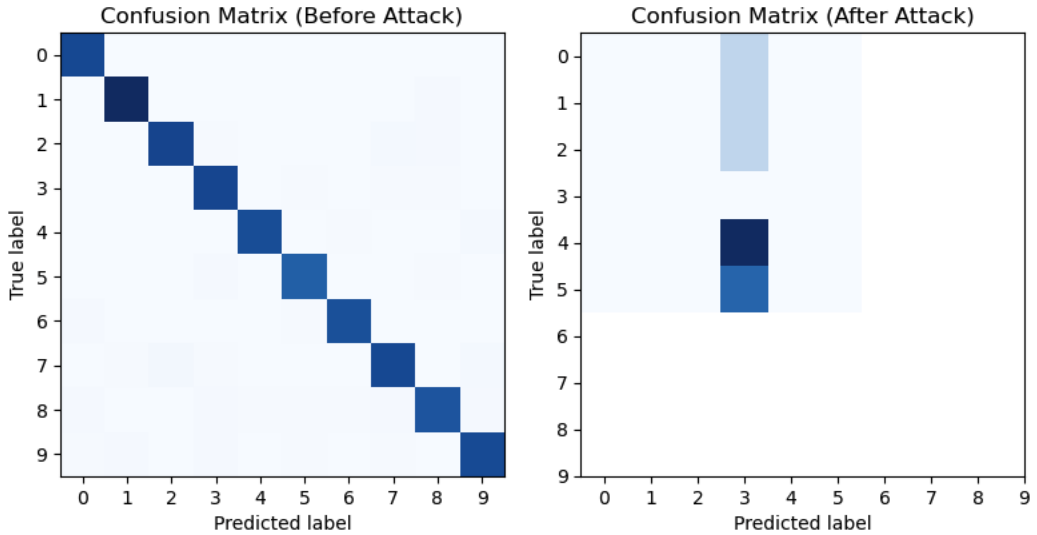
Here's a step-by-step breakdown of the CW attack using math notation:

1. Given an input image x, its true label y_{true}, a targeted class y_{target} (optional), and a constant c controlling the trade-off between perturbation magnitude and misclassification confidence:
2. Initialize the adversarial example x_{adv} as a copy of the original input x.
3. Initialize a binary search range for the perturbation magnitude: $[0, \text{upper\_bound}]$. The upper bound can be set based on the desired maximum perturbation.
4. Repeat until convergence (e.g., the maximum number of iterations reached or desired confidence achieved):
   A. Perform a change of variables to ensure the perturbation lies within the valid range. For example, use $w = \frac{1}{2}(\tanh^{-1}(2x_{\text{adv}} - 1) + 1)$ to map x_{adv} to the range $[0, 1]$.
   B. Define the objective function that must be minimized to generate the adversarial example. It typically consists of two components: the magnitude of the perturbation and the misclassification confidence. The objective function is formulated as follows:
   $\text{minimize} \quad |w - \frac{1}{2}(\tanh^{-1}(2x - 1) + 1)|^2 + c \cdot f(x\text{adv}, y_{\text{true}})$
   C. Solve the optimization problem using an optimization algorithm (e.g., gradient descent) to find the adversarial example x_{adv} that minimizes the objective function.
   D. Perform the inverse change of variables to map w back to the original input space.
   E. Clip the adversarial example to ensure it stays within a valid range (e.g., pixel range of [0, 1] for images).
   F. If the adversarial example is misclassified, update the binary search range to explore smaller perturbation magnitudes.
   G. If a targeted attack is desired and the adversarial example is classified as the targeted class, update the binary search range to explore larger perturbation magnitudes.
5. Return the final adversarial example x_{adv}.

C&W Code Example

```
adv_x_test_tmp = carlini_wagner_l2(model, tmp_input.reshape((1,28,28,1)),
                                   targeted=True, y=[TARGET_CLASS],
                                   batch_size=128, confidence=100.0,
                                   abort_early=True, max_iterations=500,
                                   clip_min=0.0,clip_max=1.0)
```





Mitigation Methods

1. **Adversarial training:** This involves training the model with both clean and adversarial examples in order to improve the model's robustness against adversarial attacks.
2. **Defensive distillation:** This involves training a larger and more complex model to generate softened probabilities that are more difficult to attack.
3. **Randomization:** This involves adding random noise to the input data, model parameters or features to make it harder for an attacker to construct effective attacks.
4. **Gradient masking:** This involves hiding or obscuring the model's gradients to prevent attackers from exploiting them for crafting adversarial examples.
5. **Feature squeezing:** This involves reducing the dimensionality of input features to make the decision boundaries smoother, thus making it harder for an attacker to craft effective adversarial examples.
6. **Ensemble methods:** This involves combining multiple models or classifiers to improve the overall performance of the system and increase robustness against adversarial attacks.
7. **Input preprocessing:** This involves pre-processing the input data to remove potentially adversarial content, such as compression artifacts, noise, or perturbations.
8. **Detection and rejection:** This involves monitoring the inputs and outputs of the model for signs of adversarial activity and rejecting any inputs that are deemed suspicious or harmful.

Adversarial Training

The process of adversarial training can be defined as follows: Given a dataset $D = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ of n training examples, where x_i represents the input image and y_i represents the true label of the image, the goal is to train a classifier $f(x; \theta)$ with parameters θ that minimizes the following objective function:

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L\left(f(x_i + \delta; \theta), y_i\right)$$

where $\mathcal{L}$ is the loss function used for training, and δ is a small perturbation added to the input image x_i to create an adversarial example.

The perturbation δ is computed by solving the following optimization problem:

$$\delta = \text{argmax}|\delta|_{\infty \leq \epsilon} L(f(x_i + \delta; \theta), y_i)$$

where ϵ is a small hyperparameter that controls the magnitude of the perturbation.

During training, the adversarial examples are generated using a variety of techniques such as FGSM, BIM, or PGD. The resulting augmented dataset is used to train the classifier $f(x; \theta)$ using standard backpropagation algorithms.

Adversarial Training

1. **Dataset Preparation**:
    1. Use natural input samples (x) and their corresponding true labels (y_{true}).
2. **Generate Adversarial Examples**:
    1. Apply attack algorithms (e.g., FGSM or PGD) to perturb input samples, causing misclassification.
    2. Ensure perturbations are small and within a specific norm bound.
3. **Augment Training Dataset**:
    1. Add adversarial examples to the training dataset.
    2. Assign labels to adversarial examples, either the true labels or targeted classes as needed.
4. **Train the Model**:
    1. Use the augmented dataset, including both natural and adversarial examples, for model training.
    2. Employ standard machine learning techniques for training.
5. **Repeat Training**:
    1. Conduct multiple training iterations to enhance the model's robustness.
6. **Evaluate the Model**:
    1. Test the model's performance on a separate test set.
    2. Measure accuracy and robustness against both natural and adversarial inputs.
    3. Compare performance before and after adversarial training to assess the defense's effectiveness.

Adversarial Training – Code Example

```python
# Define the adversarial training method
def adversarial_training(x, y, model, epochs, epsilon, batch_size):
    for epoch in tqdm(range(epochs)):
        for batch in tqdm(range(0, len(x), batch_size), leave=False):
            x_batch = x[batch:batch+batch_size]
            y_batch = y[batch:batch+batch_size]
            # Generate adversarial examples using FGSM and BIM attacks
            perturbation_fgsm = fast_gradient_method(model, x_batch, eps=epsilon, norm=np.inf, targeted=False)
            perturbation_bim = basic_iterative_method(model, x_batch, eps=epsilon, eps_iter=0.01, nb_iter=10,
                                                      norm=np.inf, targeted=False,
                                                      sanity_checks=False)

            # Combine the original image with the adversarial perturbation
            x_batch_fgsm = x_batch + perturbation_fgsm
            x_batch_bim = x_batch + perturbation_bim

        # Train the model on the original image and the adversarial example
        loss_fgsm = model.train_on_batch(x_batch_fgsm, y_batch)
        loss_bim = model.train_on_batch(x_batch_bim, y_batch)

        # Evaluate the model on the test set with FGSM attack
        perturbation_fgsm = fast_gradient_method(model, x_test, eps=epsilon, norm=np.inf, targeted=False)
        x_test_fgsm = x_test + perturbation_fgsm
        score_fgsm = model.evaluate(x_test_fgsm, y_test, verbose=0)
        test_loss_fgsm.append(score_fgsm[0])
        test_accuracy_fgsm.append(score_fgsm[1])
```

Defensive Distillation

Defensive distillation is a method for protecting machine learning models from adversarial attacks. The basic idea is to train a distilled model that is less sensitive to small changes in its inputs, which can be caused by an adversary attempting to fool the model.

The algorithm for defensive distillation involves training two models: a regular model and a distilled model. The regular model is trained on the original training data, while the distilled model is trained on the outputs of the regular model. The distilled model will learn to approximate the function computed by the regular model but with a smoother decision boundary that is less susceptible to adversarial examples.

Let X be the input space and Y be the output space. Let $f : X \to Y$ be the original model we want to defend, and let $\hat{f} : X \to Y$ be the distilled model. Let D be the training data and L be the loss function. Then, the algorithm for defensive distillation can be summarized as follows:

1. Train the regular model f on the training data D, using the loss function $L(f(x_i), y_i)$, where x_i is the ith input, and y_i is the corresponding output.
2. Generate a new training set D' by passing each input x_i in D through the regular model f and using the output $\tilde{y}_i = f(x_i)$ as the new training label.
3. Train the distilled model $\hat{f}$ on the training data D', using the loss function $L(\hat{f}(x_i), \tilde{y}_i)$.
4. Use the distilled model $\hat{f}$ to infer new inputs.

By training the distilled model on the outputs of the regular model, we hope to smooth out the decision boundary of the distilled model, making it more robust to small perturbations in the input. Additionally, since the distilled model is trained on the outputs of the regular model, it is less likely to make errors on adversarial examples crafted to fool the regular model.

Defensive distillation is not foolproof and can be bypassed by clever adversaries. However, it is effective against many attacks and can be a helpful tool in a defence-in-depth strategy for protecting machine learning models.

Defensive Distillation

1. **Training the Teacher Model:**
   - Given a dataset consisting of input samples x and their corresponding true labels y_true.
   - Train a "teacher" model using the original training data and standard machine learning techniques.
   - The teacher model can be any complex model, such as a deep neural network.
   - The goal is to create a high-performing model that can provide accurate predictions on the training data.
2. **Softening the Output Probabilities:**
   - After training the teacher model, apply a temperature parameter T to the output probabilities of the teacher model.
   - Soften the output probabilities by applying the softmax function to the logits of the teacher model divided by the temperature T.
   - Softening the probabilities smooths out the decision boundaries of the model, making it less confident and more uncertain about its predictions.
3. **Training the Student Model:**
   - Generate a new dataset of "soft" labels by using the softened output probabilities of the teacher model as the labels for the original training data.
   - Train a "student" model using the new dataset and standard machine learning techniques.
   - The student model can be a simpler model architecture compared to the teacher model.
   - The goal is to train the student model to mimic the behavior of the teacher model in terms of its softened predictions.
4. **Evaluating the Trained Student Model:**
   - Assess the performance of the trained student model on a separate test set or deployment scenarios.
   - Measure the model's accuracy and robustness against both natural inputs and adversarial examples.
   - Compare the performance of the student model to that of the teacher model and evaluate the effectiveness of the defense.

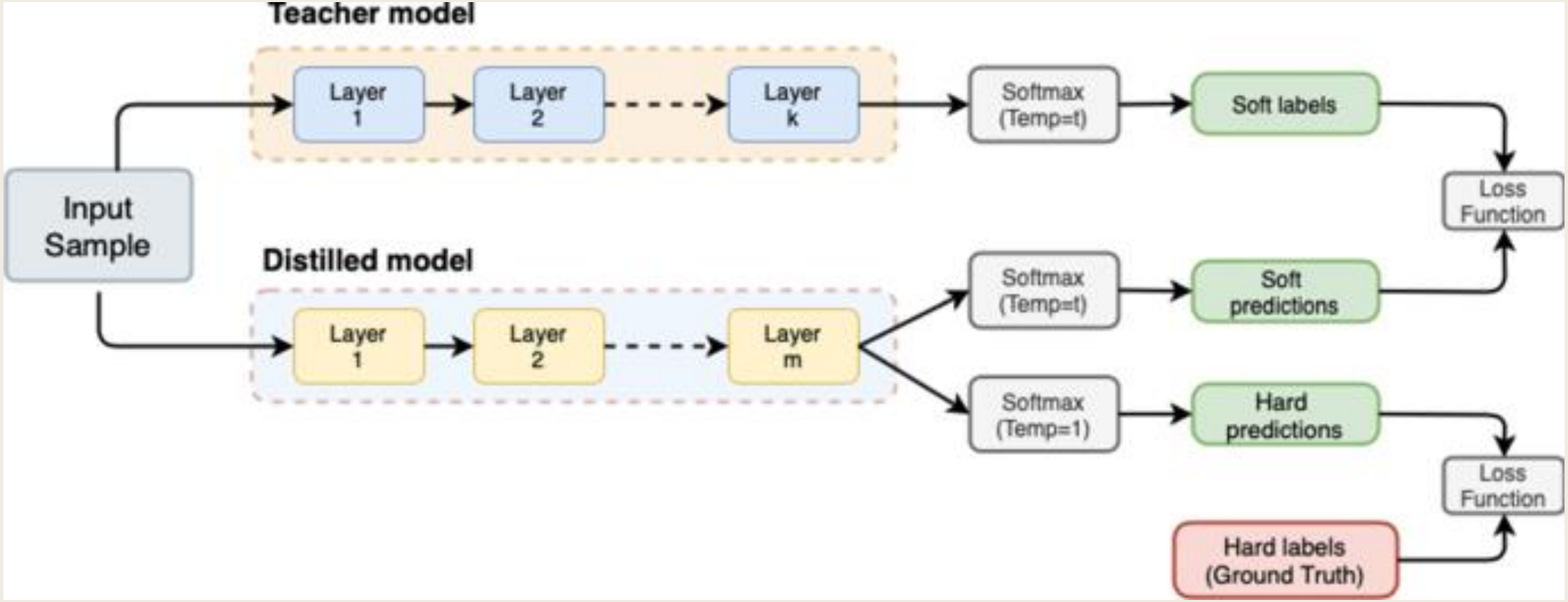
Defensive Distillation works by introducing **uncertainty** into the decision-making process of the model. By softening the output probabilities, the model becomes less confident in its predictions and is less likely to make drastic changes in response to small input perturbations. This can make the model more robust against adversarial attacks that rely on exploiting these small perturbations to cause misclassification.

Defensive Distillation

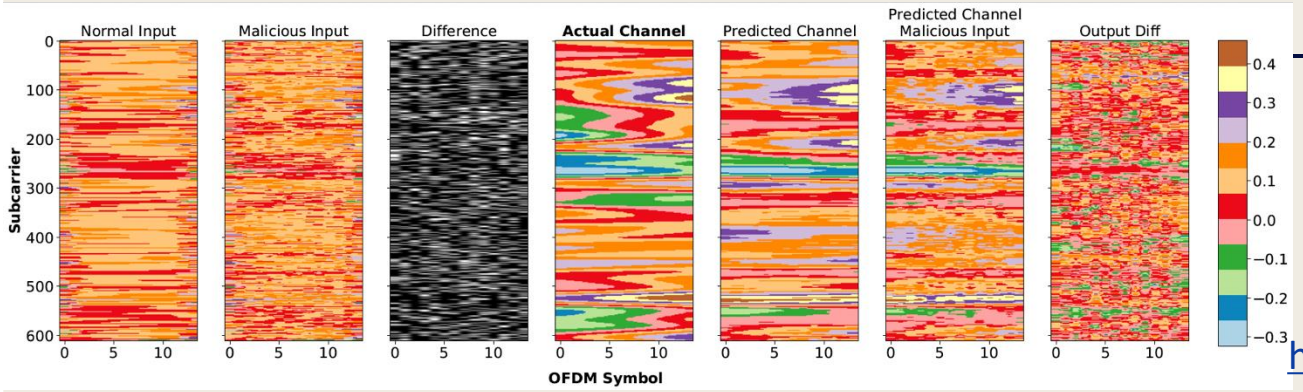
Defensive distillation is a technique used to make machine learning models more robust against adversarial attacks. Here's how it works, in simple terms:

- **Training the Original Model:**
  - First, you train a regular machine learning model (like a neural network) on your dataset. This model learns to make predictions based on the input data.
- **Creating "Soft" Labels:**
  - Instead of using the hard labels (like 0 or 1), the model produces "soft" labels, which are probabilities that indicate how confident the model is about each class. For example, instead of just saying "this is a cat," the model might say "there's an 80% chance this is a cat, 15% chance it's a dog, and 5% chance it's a rabbit."
- **Training a New Model with Soft Labels:**
  - You then use these soft labels to train a new model. This second model learns not just the final decision but the relative confidence of the model for each possible class.
- **Improved Robustness:**
  - This process of using soft labels helps the new model to generalize better and smoothens the decision boundaries. As a result, the model becomes more resistant to small, adversarial changes in the input data that are designed to fool it.

Defensive Distillation

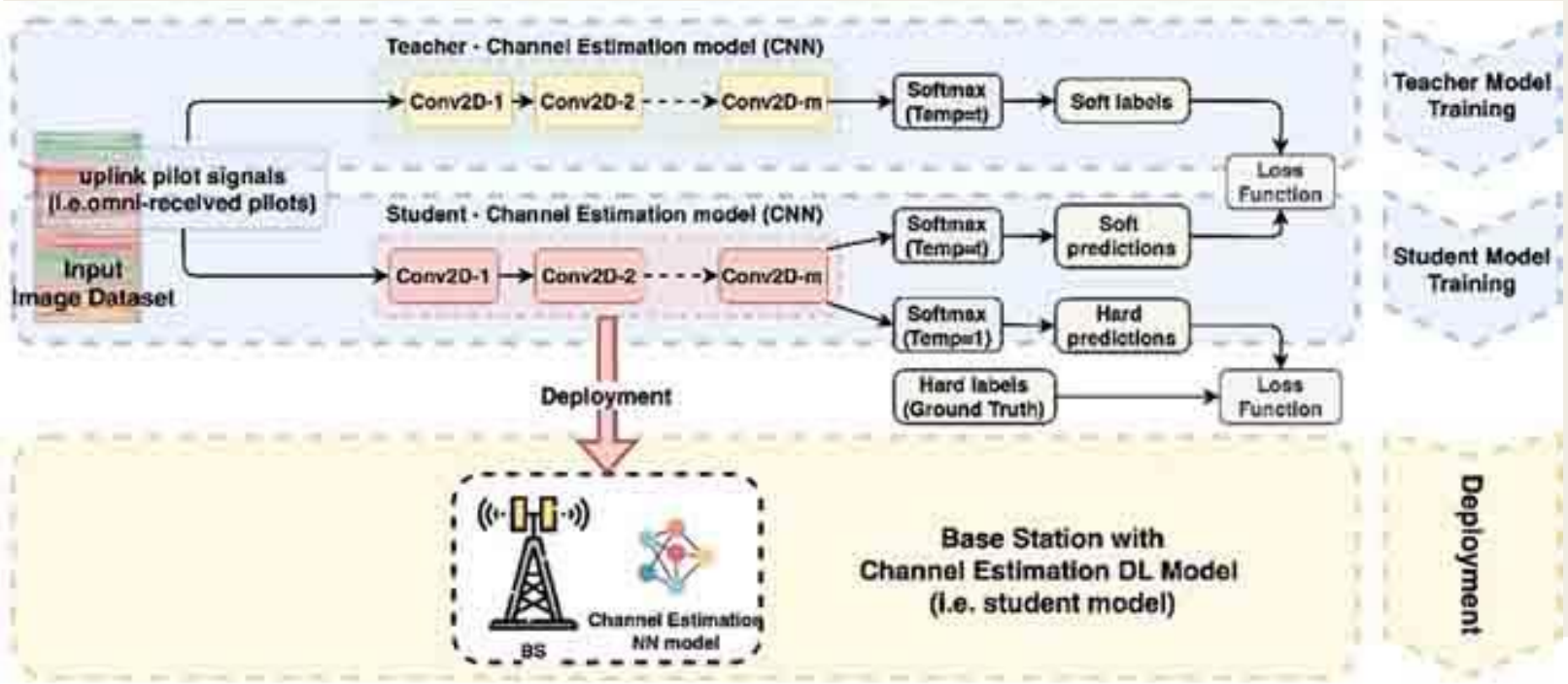


R1: Defensive Distillation for Adversarial Attack Mitigation in Wireless Networks



Catak, F. O., Kuzlu, M., Catak, E., Cali, U., & Guler, O. (2022). Defensive distillation-based adversarial attack mitigation method for channel estimation using deep learning models in next-generation wireless networks. *IEEE Access*, *10*, 98191-98203.

https://github.com/ocatak/6g-channel-estimation-dataset



Randomization based Mitigation

To mitigate such attacks, randomization-based methods add some random noise to either the input data or the model parameters. For example, during training, we can add random noise to the input data x_i, such that $x_i' = x_i + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is a random vector sampled from a Gaussian distribution with mean 0 and variance σ^2. Alternatively, we can add random noise to the model parameters θ themselves, such that $\theta' = \theta + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

The intuition behind randomization-based methods is that the added randomness can make it harder for an attacker to create effective adversarial examples. If the attacker does not know the distribution of the added noise, they cannot easily compute the perturbation δ that leads to a misclassification. Moreover, by training the model with random noise, we can encourage it to be more robust to small perturbations, which can further improve its resistance to adversarial attacks.

Formally, let $\hat{g}$ be a randomized classifier that maps an input x to a label y by adding some random noise ϵ, such that $\hat{g}(x) = \arg\max_{y \in \mathcal{Y}} f(x, \theta + \epsilon)$, where $f(x, \theta)$ is the deterministic classifier that maps x to y using the model parameters θ. Then, we can train $\hat{g}$ using the following objective:

$$L(\hat{g}) = \frac{1}{n} \sum_{i=1}^{n} E_{\epsilon_i \sim N(0,\sigma^2)}[\ell(\hat{g}(x_i + \epsilon_i), y_i)] + \lambda R(g^*)$$

Here, ℓ is a loss function that measures the discrepancy between the predicted label and the true label, $R(\hat{g})$ is a regularization term containing some prior knowledge about the distribution of the added noise, and λ is a hyperparameter that controls the strength of the regularization.

The first term in the objective encourages the classifier to make accurate predictions on the original data points x_i, while the second term encourages it to be robust to small perturbations by penalizing large changes in the model parameters θ. By training the classifier with this objective, we can effectively balance the trade-off between accuracy and robustness.

Randomization based Mitigation

```
# Define the loss function with randomization-based mitigation
def loss_fn(y_true, y_pred):
    # Randomization parameters
    epsilon = 0.1
    sigma = 0.01

    # Perturb the input data
    n = tf.shape(y_true)[0]
    noise = tf.random.normal(shape=(n, 28, 28), mean=0.0, stddev=sigma)
(1) perturbed_x = tf.slice(x_train, [0, 0, 0], [n, 28, 28]) + epsilon * noise

    # Evaluate the model with the perturbed input data
(2) y_pred_perturbed = model(perturbed_x)

    # Calculate the cross-entropy loss
(3) ce_loss = tf.keras.losses.categorical_crossentropy(y_true, y_pred, from_logits=False)

(4) # Calculate the KL divergence between the original and perturbed predictions
    kl_div = tf.nn.softmax_cross_entropy_with_logits(y_pred, y_pred_perturbed)

(5) # Return the combined loss
    return tf.reduce_mean(ce_loss + kl_div)
```

Refer to Jupyter Notebook's numeric example

Gradient Masking

4.4 Gradient Masking

The Gradient Masking method is a type of adversarial defense method that aims to mitigate adversarial examples by "masking" the gradients of the mode with respect to the input. It is also known as Gradient Obfuscation or Jacobian Masking. The idea behind this method is to add random noise to the gradients of the model during the training phase to make it harder for an attacker to construct adversarial examples.

Let's consider a neural network model f_θ with input x and output y, where θ denotes the model parameters. The goal of the attacker is to find a small perturbation δ to the input x such that the output $f_\theta(x+\delta)$ is misclassified. In other words, the attacker wants to solve the following optimization problem:

$$\begin{aligned} &\underset{\delta \in \mathbb{R}^n}{\text{minimize}} \quad |\delta| p \\ &\text{subject to} \quad f\theta(x+\delta) = t \end{aligned}$$

where t is the target label and p is a norm. The gradient-based attack methods, such as Fast Gradient Sign Method (FGSM) and Basic Iterative Method (BIM), use the gradients of the model with respect to the input to compute the perturbation δ that maximizes the loss function. Therefore, by adding random noise to the gradients, the Gradient Masking method aims to reduce the effectiveness of these attacks.

To implement Gradient Masking, we add a random noise r to the gradients of the model during the training phase. Let $L(\theta)$ be the loss function of the model, and let $J(x)$ be the Jacobian matrix of f_θ with respect to x. The masked gradient $\nabla_x L(\theta)$ is computed as follows:

$$\nabla_x L(\theta) = J(x)^T \cdot g + r$$

where g is the gradient of the loss function with respect to the output y. The random noise r is sampled from a Gaussian distribution with zero mean and small standard deviation σ.

During the training phase, the masked gradient is used to update the model parameters θ using a gradient descent algorithm. At the inference phase, the model is evaluated on the test data without any gradient masking.

The Gradient Masking method aims to make it harder for an attacker to compute the perturbation δ by adding random noise to the gradients of the model during the training phase. This method can be effective against gradient-based attacks, but it may not be robust against other types of attacks.

This code demonstrates the training of a neural network model on the MNIST dataset using a custom loss function that combines cross-entropy loss and KL divergence. The gradient masking technique is used to modify the gradients during backpropagation, potentially providing additional robustness to

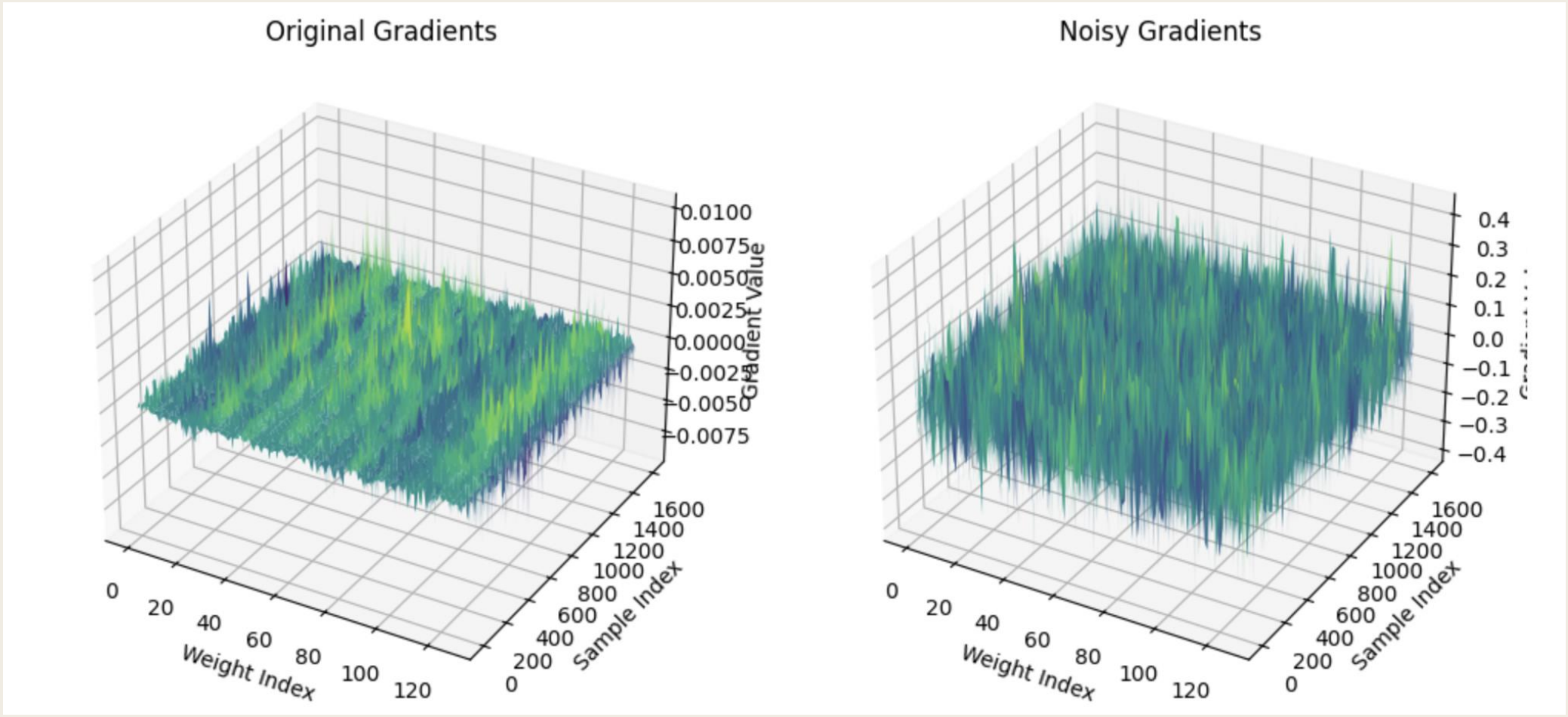
Gradient Masking

- **Adding Noise to Gradients:**
  - During the training of the model, you add random noise to the gradients.
  - This noise makes the directions (gradients) less accurate, confusing the attacker's process of finding the right perturbation to fool the model.

Example:

- **Training Phase**:
  - You have a model that classifies digits. During training, instead of using clean gradients, you add a small amount of random noise to them.
  - Imagine you want to move a small ball up a hill using directions. If the directions are accurate, you move straight up. If there's noise in the directions, your path becomes wobbly and less efficient.
- **Using Masked Gradients**:
  - The model learns with these noisy (masked) gradients, making it harder for anyone to predict exactly how to perturb the input to change the output.
- **Inference Phase**:
  - During inference (actual use of the model), the model is evaluated normally without adding noise to the gradients.
  - However, because the model was trained with noisy gradients, it has become less sensitive to small perturbations, making it more robust against adversarial attacks.
- **Key Takeaways**
  - **Gradient Masking**: Adding noise to the gradients during training to make it harder for attackers to use gradient-based methods to generate adversarial examples.
  - **Effectiveness**: Makes the model more robust to small perturbations and gradient-based attacks.
  - Training: Involves modifying the gradient calculation process to include random noise.

Gradient Masking



Comparison

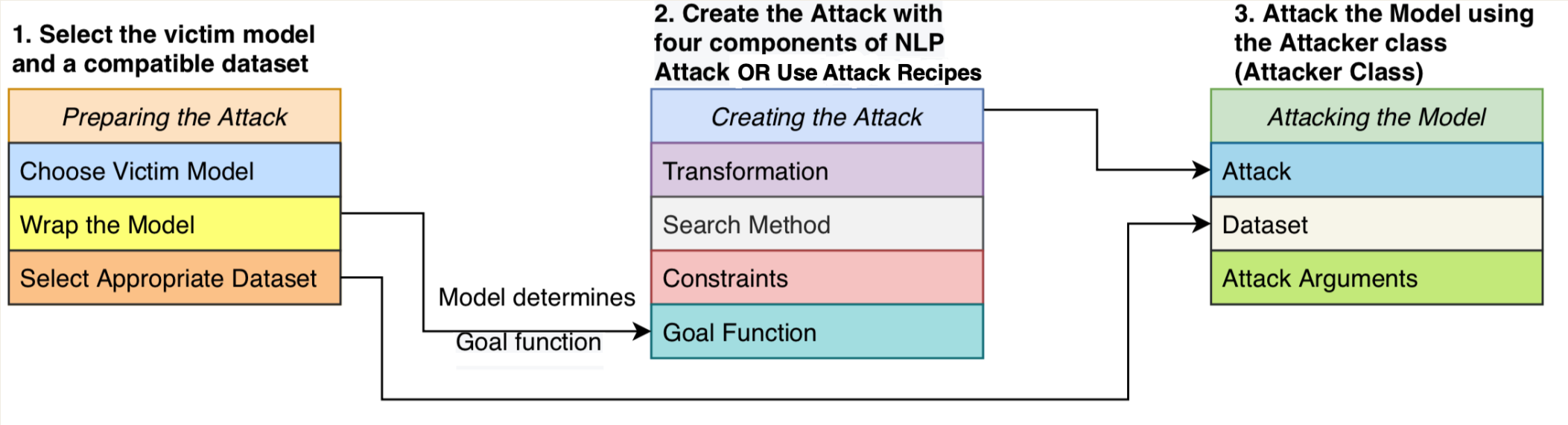
Aspect	Gradient Masking	Randomization-Based Mitigation
Objective	Obfuscate gradients to mislead attackers	Introduce randomness to inputs or parameters
Mechanism	Alters or hides gradient information	Adds random noise to data or model parameters
Impact on Attackers	Makes gradient-based attacks harder to perform	Reduces the effectiveness of precise adversarial examples
Common Techniques	Non-linear activations, gradient clipping, saturation	Input perturbation, parameter perturbation, randomized smoothing
Example Code Snippet	Adding noise to gradients	Adding noise to inputs or loss functions

- **Gradient Masking**: Focuses on altering or obfuscating gradient information to make it difficult for attackers to generate adversarial examples. It directly manipulates the gradient calculations.
- **Randomization-Based Mitigation**: Introduces randomness in the training or inference process, typically by adding noise to inputs or model parameters. It aims to disrupt the attacker's ability to consistently generate effective adversarial perturbations.

TextAttack: An Adversarial Attack Framework for NLP

- What is TextAttack?
  - A Python framework for generating adversarial examples in NLP.
  - Allows evaluation and improvement of model robustness.
  - Provides tools for attack, defense, and data augmentation.
- https://textattack.readthedocs.io/en/master/
- Key Features:
  - **Comprehensive Attack Methods**: Supports multiple attack strategies like gradient-based, search-based, and heuristic attacks.
  - **Pre-trained Models**: Includes a variety of pre-trained models for easy testing (e.g., lstm-mr).
  - **Flexible Transformations**: Allows for different word-level and sentence-level transformations.
  - **Custom Constraints**: Ensures generated adversarial examples are valid and meaningful.
  - **Extensible Framework**: Easily integrates with custom models and datasets.

TextAttack



- **Why Use TextAttack?**
  - Test and improve model robustness.
  - Identify vulnerabilities in NLP models.
  - Enhance model reliability and security.
- **Applications**:
  - Sentiment analysis, text classification, machine translation, and more.
  - Research in adversarial attacks and defenses.