

Adversarial Machine Learning

BBVA
Next Technologies

Índice general

1	Introducción	1
1.1	Breve introducción al machine learning	1
1.2	¿Qué es el Adversarial Machine Learning?	2
1.3	Taxonomía de ataques	3
2	Ataques de extracción	9
2.1	Técnicas de ataque	10
2.2	Medidas defensivas	15
3	Ataques de inversión	20
3.1	Causas del filtrado de información	21
3.2	Técnicas de ataque	22
3.3	Medidas defensivas	24
3.4	Herramientas	25
4	Ataques de envenenamiento	27
4.1	Técnicas de ataque	27
4.2	Medidas defensivas	34
4.3	Herramientas	35
5	Ataques de evasión	37
5.1	Causas de los ejemplos adversarios	38
5.2	Técnicas de ataque	38
5.3	Medidas defensivas	44
5.4	Herramientas	47
6	Tutorial: Adversarial Robustness Tool (ART)	51
6.1	Prerrequisitos e instalación	51
6.2	iHola, ART!	52
6.3	Ataques de evasión	54
6.4	Ataques de inversión	58
6.5	Ataques de envenenamiento	61
6.6	Ataques de extracción	67
7	Futuro y conclusiones	70
Referencias		73
A Resumen de herramientas opensource		75

1 Introducción

El *machine learning* (ML) y el *deep learning* (DL) han ganado una gran popularidad en los últimos tiempos tanto en el mundo académico como en el sector empresarial debido a las **numerosas aplicaciones** tales como clasificación de imágenes y vídeo, reconocimiento e identificación facial, detección de malware, procesamiento del lenguaje natural o sistemas de conducción autónoma. Tanto el uso del *machine learning* como del *deep learning* han mejorado las técnicas anteriores.

Este auge ha sucedido gracias al **aumento del poder de cómputo de las tarjetas gráficas** (GPUs), principalmente debido a las plataformas cloud que ofrecen GPUs bajo demanda a un precio asequible, un recurso muy necesario en el caso del *deep learning*. Otro factor que ha favorecido esta explosión del ML/DL ha sido el **aumento del consumo y generación de datos**, cada vez mayor; se espera que en los próximos años siga creciendo a un ritmo incluso más rápido debido a la transformación digital de las empresas y la normalización de los dispositivos IoT. Todo esto propicia que sea más sencillo entrenar modelos en un margen razonable de tiempo y con un **coste asumible**.

Los modelos mejoran cada día y son capaces de tomar mejores decisiones sin la supervisión de un humano. En aplicaciones críticas, **la seguridad de los modelos se vuelve algo indispensable si estos modelos tienen la capacidad de tomar decisiones que pongan en riesgo la vida de las personas**. El caso más actual y más cercano a llegar al público son los sistemas de conducción autónoma que *basan su funcionamiento en deep learning*, y la seguridad de estos modelos es fundamental que se audite de forma rigurosa antes de ser puestos en producción. Estas auditorías requieren de muchas pruebas y el Adversarial Machine Learning es el primer paso en la estandarización de las metodologías necesarias para contar con un marco de pruebas robusto que garantice el buen funcionamiento del modelo y la seguridad de sus usuarios.

1.1 Breve introducción al machine learning

El *machine learning* es la disciplina de la Inteligencia Artificial relacionada con la implementación de algoritmos que pueden aprender de forma autónoma. En ella, se incluye el *deep learning*, que diseña e implementa algoritmos, conocidos como redes neuronales, inspirados en la estructura y funciones del cerebro humano.

Los algoritmos de *machine learning* se pueden clasificar en tres grupos dependiendo de la salida que produzcan, que reciben el nombre de tipos de aprendizaje:

- El **aprendizaje supervisado** consiste en inferir una función a partir de da-

tos de entrenamiento etiquetados, es decir, para cada uno de los datos se tiene tanto la entrada como la salida esperada.

- El **aprendizaje no supervisado** consiste en inferir una función a partir de datos de entrenamiento no etiquetados, es decir, sólo se conoce la entrada de cada uno de los datos.
- El **aprendizaje por refuerzo** se preocupa por cómo los agentes de software deben tomar acciones en un entorno para maximizar algún tipo de recompensa acumulativa.

El proceso de obtener un modelo de *machine learning* se da durante la **fase de entrenamiento** y consiste en minimizar una función de error (denominada función de pérdida, *loss function* en inglés) ajustando una serie de parámetros. A los parámetros que no se ajustan mediante el entrenamiento se les llama hiperparámetros.

Una vez obtenido el modelo, podemos hacerle peticiones para obtener una predicción (o inferencia) cuya exactitud y corrección variará en gran medida en función de cómo se haya realizado el entrenamiento. Esto es lo que se llama **fase de inferencia**.

Pese a la gran variedad de usos que se le está dando a esta tecnología, los modelos de *machine learning* son entrenados para realizar alguno de estos dos tipos de **tareas**:

- **Clasificación**: si las salidas del modelo toman un número finito de valores. A las salidas se les llama etiqueta. Por ejemplo, si queremos distinguir entre dígitos manuscritos (0, 1, ..., 9), las salidas posibles del modelo son 10, que se corresponden con los 10 dígitos disponibles.
- **Regresión**: si las salidas del modelo toman un número infinito de valores. Por ejemplo, predecir la temperatura que hará mañana en un lugar determinado (la salida toma infinitos valores).

1.2 ¿Qué es el Adversarial Machine Learning?

En una pieza de software las vulnerabilidades suelen tratar sobre errores de configuración, valores por defecto, errores en el código o vulnerabilidades en las dependencias. Sin embargo, al desplegar un modelo de ML/DL debería ser esperable que los modelos no presentaran ninguna vulnerabilidad, pero esto no es cierto en la mayoría de los casos. Este es el tema central del *Adversarial Machine Learning*, que es una rama del *machine learning* que trata de averiguar qué ataques puede sufrir un modelo en la **presencia de un adversario malicioso** y

cómo protegerse de ellos.

El adversario, en función del conocimiento que tiene del modelo que pretende atacar puede realizar tres tipos de ataques:

- 1. Ataques de caja blanca:** el adversario tiene acceso a la arquitectura usada por el modelo, a los datos de entrenamiento, a los parámetros e hiperparámetros.
- 2. Ataques de caja negra:** el adversario sólo tiene acceso a las entradas y salidas del modelo.
- 3. Ataques de caja gris:** el ataque se encuentra en un punto intermedio entre los dos tipos de ataques anteriores.

1.3 Taxonomía de ataques

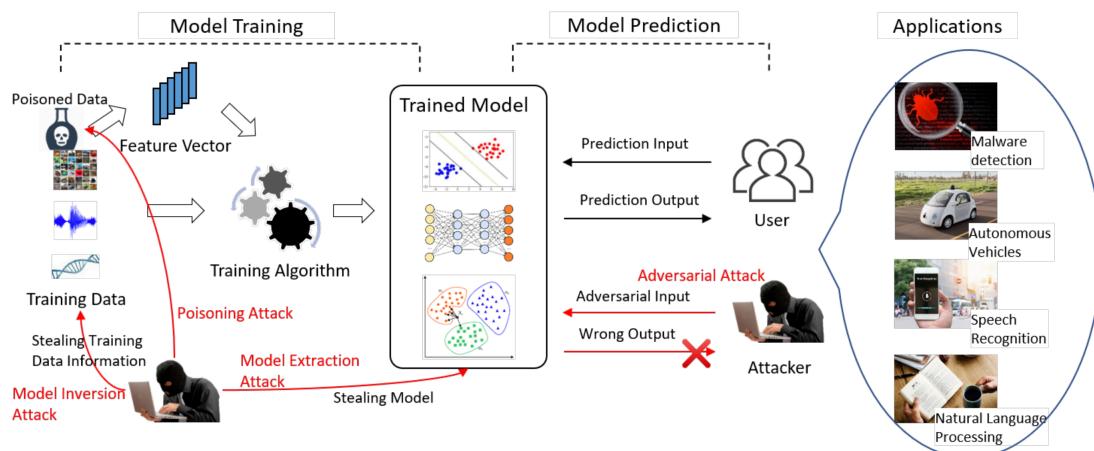


Figura 1.1: Taxonomía de los ataques del Adversarial Machine Learning. Fuente: [6].

Aunque es posible realizar numerosos ataques a los modelos de *machine learning*, el Adversarial Machine Learning clasifica todos estos los ataques en cuatro tipos:

- **Envenenamiento**
- **Evasión**
- **Extracción**
- **Inversión**

La Figura 1.1 muestra la relación entre los tipos de ataque, el adversario (o atacante) y la fase en la que se puede producir el ataque (entrenamiento o inferencia).

1.3.1 Ataques de envenenamiento

Los ataques de envenenamiento (*poisoning attacks* en inglés) también son conocidos como ataques causativos (*causative attacks* en inglés). El adversario trata de **corromper el conjunto de entrenamiento** con el objetivo de que el modelo aprendido produzca una clasificación errónea que beneficia al adversario.

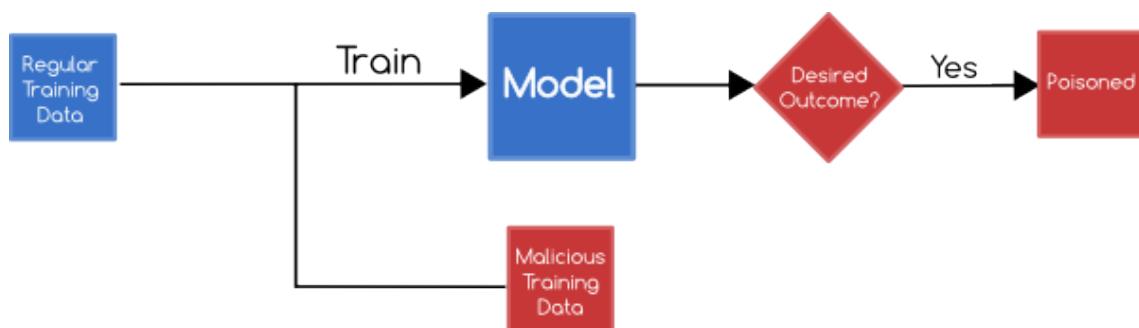


Figura 1.2: Ataque de envenenamiento. Fuente: [Sarah Jamie Lewis](#)

Este tipo de ataques se realiza en la fase de entrenamiento y es muy difícil averiguar la vulnerabilidad subyacente ya que se puede transmitir a todos los modelos que empleen estos datos. Este ataque se puede realizar tanto en caja blanca como en caja negra y **compromete la disponibilidad** de los modelos.

1.3.2 Ataques de evasión

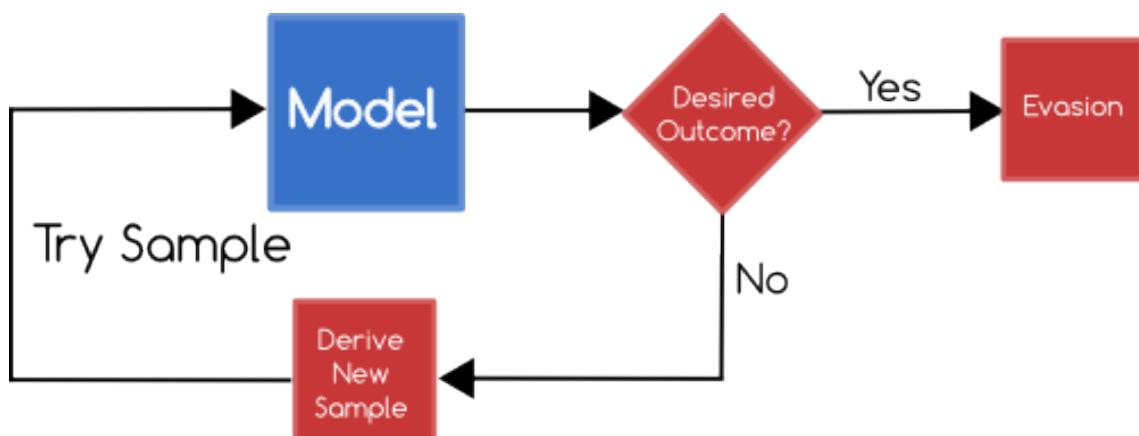


Figura 1.3: Ataque de evasión. Fuente: [Sarah Jamie Lewis](#)

Los ataques de evasión (*evasion attacks* en inglés) son también conocidos como ataques exploratorios (*exploratory attacks* en inglés). El objetivo del adversario

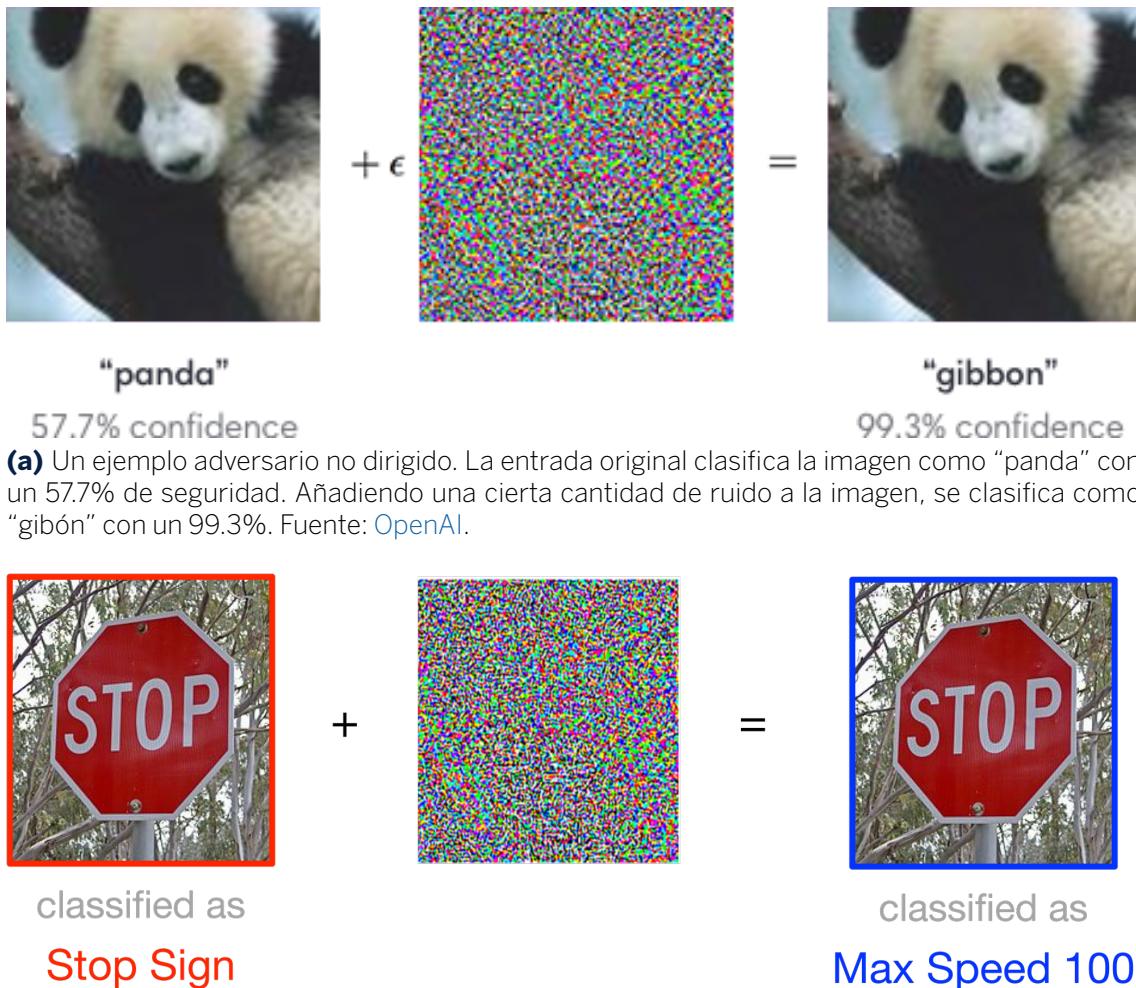


Figura 1.4: Un par de ejemplos adversarios.

es **inyectar en la entrada una pequeña cantidad de ruido** para que un clasificador prediga la salida (o etiqueta) de forma incorrecta.

La mayor diferencia entre el ataque de envenenamiento y el ataque de evasión es que, a diferencia del ataque de envenenamiento, que se realiza en la fase de entrenamiento, el ataque de evasión se realiza en la fase de inferencia.

Este tipo de ataque se realiza modificando ligeramente las entradas introduciendo una cierta cantidad de ruido. Estas entradas maliciosas reciben el nombre de **ejemplo adversario** (*adversarial example* en inglés). Se pueden crear sobre distintos tipos de datos, aunque los ejemplos adversarios más extendidos y conocidos son sobre imágenes. De hecho, los ejemplos adversarios sobre imágenes se crean para que sean **imperceptibles para el ojo humano**. Este tipo de ataques

se pueden hacer tanto de caja negra como de caja blanca y se realizan durante la fase de inferencia.

El adversario puede requerir ser más preciso a la hora de obtener una salida para obtener sus objetivos: puede necesitar una salida con una etiqueta específica o simplemente obtener un resultado no esperado. Esto permite diferenciar entre dos tipos de ataque: dirigido o no dirigido.

- **Dirigido:** el adversario quiere obtener una salida específica de su elección.
- **No dirigido:** el adversario quiere obtener un resultado distinto al esperado.

Un ataque no dirigido puede darse en el primer ejemplo donde un adversario sólo quiere que se clasifique de forma incorrecta la imagen del panda. Un ataque dirigido puede producirse en la segunda imagen donde el adversario quiere que una señal de stop se clasifique como de máxima velocidad a 100 km/h (Figura 1.4).

Los ataques dirigidos suelen ser más difíciles de obtener en la práctica que los no dirigidos y son más peligrosos desde el punto de vista del adversario, mientras que los no dirigidos son más fáciles de obtener, pero menos potentes para un adversario en la mayoría de casos.

1.3.3 Ataques de inversión



Figura 1.5: Ejemplo de ataque de inversión. A la izquierda se muestra un ejemplo reconstruido y a la derecha el dato de entrenamiento original. Fuente: [3].

Los ataques de inversión de modelos (*model inversion attacks* en inglés) consisten en que un adversario intenta aprovechar las predicciones del modelo para

comprometer la privacidad del usuario o inferir si unos datos determinados fueron empleados en el conjunto de entrenamiento o no. Se puede emplear tanto en caja blanca como en caja negra. Un ejemplo de este ataque se puede ver en la Figura 1.5.

Este tipo de ataque tiene especial relevancia entre los modelos que han sido entrenados con **datos sensibles**, como pueden ser los datos clínicos, que requieren de una protección especial.

1.3.4 Ataques de extracción

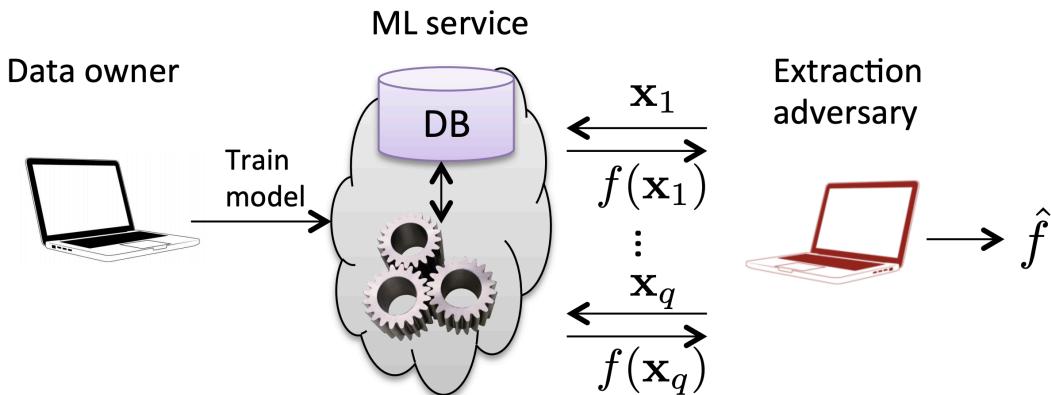


Figura 1.6: Ataque de extracción de modelos. Fuente: [15].

Los ataques de extracción de modelos (*model extraction attacks* en inglés) consisten en que un adversario intenta **robar los parámetros** de un modelo de machine learning. Este tipo de ataque permite comprometer la **propiedad intelectual y confidencialidad** de un modelo, y permite realizar ataques de evasión y/o inversión. Este ataque se puede realizar tanto en caja blanca como caja negra.

Este ataque ha sido empleado para robar la propiedad intelectual de un modelo, evitando pagar por consumir un servicio de *Machine Learning as a Service* (MLaaS) como los disponibles en AWS, GCP, Azure o BigBL, por ejemplo.

El esquema para extraer un modelo es el siguiente (Figura 1.6):

1. Un usuario legítimo entrena su modelo a partir de datos, o delega el entrenamiento en servicios MLaaS para obtener un modelo que es desplegado en algún sistema, accesible al atacante a través de una API.
2. Este último, realizando peticiones a la API, es capaz de deducir (en mayor o menor medida) un modelo equivalente al del usuario legítimo.

Tabla resumen

La Tabla 1.1 muestra un resumen con las principales características de cada uno de los tipos de ataque que propone el Adversarial Machine Learning.

Tabla 1.1: Características de los ataques de Adversarial Machine Learning

Característica	Envenenamiento	Evasión	Extracción	Inversión
Caja blanca	•	•	•	•
Caja negra	•	•	•	•
Fase de entrenamiento	•			
Fase de inferencia		•	•	•
Vulnera confidencialidad			•	
Vulnera disponibilidad	•	•		
Vulnera propiedad intelectual			•	
Compromiso datos entrenamiento				•

2 Ataques de extracción

Los ataques de extracción de modelos permiten a un adversario **robar los parámetros** de un modelo de *machine learning*. Este robo se produce al hacer peticiones al modelo objetivo con entradas preparadas para extraer la mayor cantidad de información posible, y con las respuestas obtenidas inferir (de forma aproximada) los valores de los parámetros del modelo original. Este tipo de ataque vulnera la **confidencialidad del modelo** y **compromete la propiedad intelectual** del mismo. Al disponer el adversario de un modelo equivalente al original, le permite además **realizar otros tipos de ataque** como evasión y/o inversión. Los métodos de extracción se han centrado en modelos de *deep learning* en un mayor medida debido a su mayor coste en tiempo y en datos de entrenamiento, aunque también es posible aplicarlos a otros modelos de *machine learning*. Este tipo de ataque se puede realizar conociendo ciertos detalles internos del modelo (caja blanca) o sin tener ningún conocimiento del modelo a extraer (caja negra).

Este último tipo de ataque es el más utilizado y ha sido empleado para robar la propiedad intelectual de un modelo, evitando tener que pagar por consumir un servicio de *Machine Learning as a Service* (MLaaS) como los disponibles en AWS, GCP, Azure o BigBL, por ejemplo. En este escenario, el adversario sólo tiene acceso a las entradas que introduce al modelo que intenta robar y, a las salidas que devuelve el mismo. El adversario está limitado de 3 maneras:

- 1. Conocimiento del modelo:** el adversario no conoce los detalles internos la arquitectura interna del modelo o los hiperparámetros.
- 2. Acceso a los datos:** el adversario no conoce el proceso de entrenamiento y no tiene acceso a otro conjunto de datos con la misma distribución que los originales.
- 3. Cantidad de peticiones:** el adversario puede ser detectado si hace peticiones muy frecuentemente al modelo.

La Figura 2.1 muestra el proceso para robar un modelo:

- El adversario **realiza peticiones al modelo** que intenta robar y obtiene peticiones al modelo que intenta robar y obtiene las salidas correspondientes del mismo.
- El adversario **usa las entradas y salidas** para extraer el modelo, eligiendo la aproximación que le convenga al adversario en su escenario.
- El **adversario roba los datos confidenciales** entre los que se incluyen los parámetros, hiperparámetros, arquitectura, fronteras de decisión y/o funcionalidad del modelo.

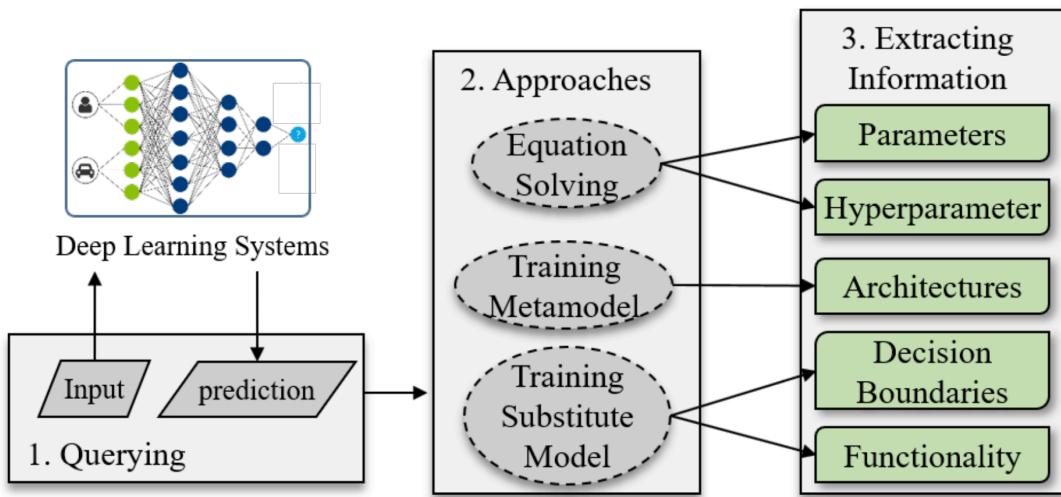


Figura 2.1: Proceso de extracción de modelos. Fuente: [6].

2.1 Técnicas de ataque

Según [11] existen 3 aproximaciones para extraer modelos: **resolución de ecuaciones**, entrenar un **metamodelo** y entrenar un **modelo sustituto**.

2.1.1 Resolución de ecuaciones

Uno de los métodos más sencillos, y que da bastante buenos resultados es el “resolución de ecuaciones”, donde **conocer ciertos datos del algoritmo permite robar los parámetros del modelo**, y con ello, el modelo. Este tipo de ataque se ha realizado con éxito en [15] y [16], entre otros. En el primero, se aplica para robar los parámetros del modelo, y el segundo para robar los hiperparámetros del mismo. Veamos un ejemplo de cómo realizar este tipo de ataque: supongamos que conocemos el tipo de algoritmo usado. Algunos de los servicios MLaaS nos dan esta información, como [Amazon Machine Learning](#). Supongamos que conocemos que el algoritmo utilizado es una [regresión logística](#). Así pues, sabemos que

$$f(x) = \frac{1}{1 + e^{-\beta x}}$$

Así pues, si x es un vector de dimensión n , $n + 1$ peticiones linealmente independientes son suficientes para obtener los parámetros del modelo (β), y con ello, robar el modelo. Para conseguirlo, se hacen $n + 1$ peticiones, con lo que se consiguen $n + 1$ pares de la forma

$$(x, f(x)).$$

Este sencillo ataque tiene **consecuencias devastadoras en servicios MLaaS**, pues consigue robar modelos con pocas consultas y un coste ínfimo. Un caso de uso interesante de este tipo de ataques que se destaca en [15] es el de los servicios MLaaS caja negra, donde un usuario sube sus datos de entrenamiento y el servicio es capaz de entrenar un modelo, pero el usuario no conoce los parámetros y sólo puede acceder al modelo a través de una API, como si de una caja negra se tratase, y pagar por cada petición realizada. Con el ataque anterior se conseguiría entrenar un modelo transparente para el usuario (normalmente, bastante costoso) de forma que al acabar de entrenar, con muy pocas consultas robar el modelo, evitando el coste posterior de estos servicios.

Este ataque se ha realizado tanto para regresión logística (para clasificación binaria como para multiclase) como con **perceptrones multicapa**.

El método de resolución de ecuaciones también ha sido aplicado para el robo de hiperparámetros. Estos parámetros son **más costosos computacionalmente** de obtener que los parámetros del entrenamiento. En [16] se aplica un método similar al anterior para el robo de los hiperparámetros. Los autores del artículo parten de que muchos de los algoritmos de *machine learning* tratan de minimizar una función objetivo de la forma

$$f(w) = L(x, w) + \lambda \cdot R(w)$$

donde, f es la función objetivo; L , la función de pérdida; λ , el hiperparámetro; y R , término de regularización. Así pues, el objetivo es obtener el valor de λ . En [16], sólo se centran las funciones objetivo con la forma de f , pero destacan que es conveniente investigar robar otros hiperparámetros como K en KNN, la arquitectura utilizada, la tasa de Dropout o el tamaño de *minibatch* utilizado, en el caso de las redes neuronales.

Para robar el parámetro λ , los autores hacen uso de una observación que será clave para el método propuesto: puesto que el objetivo es minimizar la función f , un mínimo en esa función, implica que los puntos cercanos al mínimo, tendrán valores mayores que el propio mínimo. En términos matemáticos, un mínimo vendrá dado con la anulación del gradiente, esto es, al minimizar la función, el gradiente del valor mínimo valdrá 0. Esto es,

$$\nabla f = 0.$$

Se utilizará esta observación para robar el hiperparámetro. Así pues, **el objetivo es minimizar el gradiente de la función de pérdida** e igualar a 0. De esta manera, se obtiene un sistema de ecuaciones lineales, cuya incógnita es λ .

El sistema de ecuaciones es sobre determinado, es decir, con más ecuaciones que incógnitas. Por lo que es necesario aplicar una solución aproximada, por ejemplo, utilizando el método de **mínimos cuadrados** para obtener λ .

Para exemplificar, consideremos el caso de la [regresión contraída](#) (o *ridge regression* en inglés). Su función objetivo viene dada por

$$f(w) = \|y - X^T w\|_2^2 + \lambda \cdot \|w\|_2^2$$

El gradiente de la función f viene dado por

$$\frac{\partial f}{\partial w} = -2Xy + 2XX^T w + 2\lambda w$$

El parámetro λ viene dado (a consecuencia de resolver por mínimos cuadrados) por

$$\lambda = -(a^T a)^{-1} a^T b$$

con

$$a = w \text{ y } b = X(X^T w - y).$$

Este método ha sido aplicado a distintos tipos de **regresión, regresión logística, SVM y redes neuronales**.

En [16] se explica cómo aplicar este método para funciones no diferenciables y cómo solucionar la no diferenciabilidad de las mismas. Además, se explica cómo aplicar el método a algoritmos que emplean un kernel.

2.1.2 Entrenamiento de un metamodelo

Metamodel es un clasificador propuesto en [11], que haciendo peticiones a un modelo de clasificación sobre las salidas Y para ciertas entradas X , el adversario entrena un modelo

$$F^m: Y \rightarrow X$$

Este método se ha empleado para **obtener atributos del modelo** como arquitectura del modelo, tiempo de operación y tamaño del conjunto de entrenamiento.

2.1.3 Entrenamiento de un modelo sustituto

El método más empleado para robar los modelos como caja negra (típicamente modelos de *deep learning*) consiste en un modelo sustituto, es decir, **un modelo que imite al modelo original que se pretende extraer**. La forma de funcionar de este ataque descrita en [17] es la siguiente (Figura 2.2):

1. El adversario **obtiene respuestas** del modelo a través de datos de entrada.

2. El adversario **observa etiquetas devueltas** por el clasificador.
3. El adversario **usa los datos de entrada y las etiquetas para entrenar un modelo de deep learning y optimizar sus hiperparámetros**. El modelo resultante infiere todos los aspectos del modelo original como datos de entrenamiento, tipo de clasificador e hiperparámetros del clasificador.

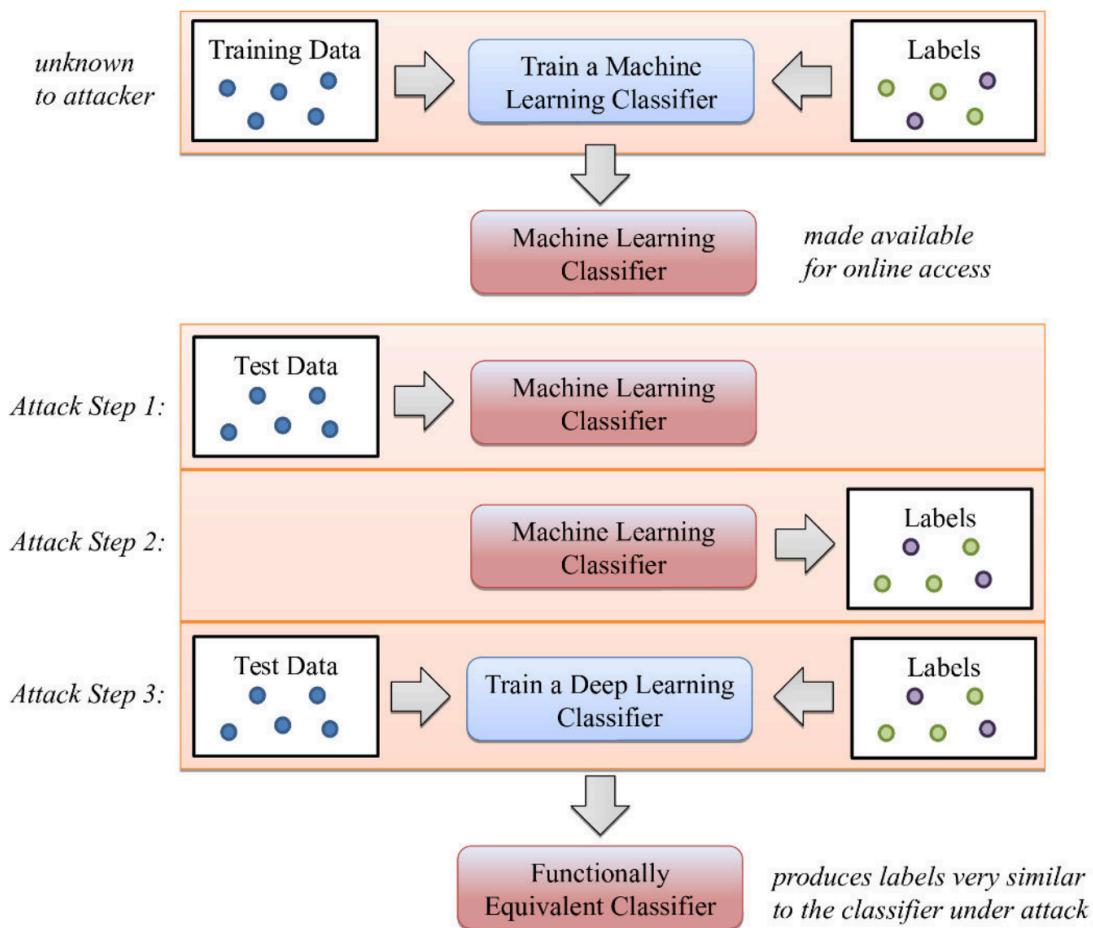


Figura 2.2: Método del modelo sustituto. Fuente: [17].

Este tipo de ataque se llevó a cabo sobre [Support Vector Machine \(SVM\)](#), [Näive Bayes](#) y redes neuronales sobre el conjunto de datos de [Reuters-21578](#). De los resultados obtenidos en [17], los autores dedujeron que **los modelos de deep learning** (empleados como modelos sustitutos) **son capaces de producir modelos muy parecidos a los originales**. Sin embargo, el recíproco no es cierto: **otros clasificadores no pueden imitar la funcionalidad de un modelo de deep learning**.

Otra variante de este ataque se puede encontrar en [7]. La diferencia sustancial con el método anterior es que en este método se dispone de un **número máximo**

de peticiones que se pueden realizar al modelo que el atacante quiere robar. Este método tiene seis pasos:

- 1. Selección de los hiperparámetros del modelo:** se selecciona una arquitectura y los hiperparámetros del modelo sustituto. La extracción de hiperparámetros se puede hacer en forma de caja negra, aplicando los ataques de [16].
- 2. Colección de datos iniciales:** se construye un conjunto de datos iniciales sin etiquetar que forman la base de la extracción. Son seleccionados en concordancia con las capacidades del adversario.
- 3. Consultas al modelo:** todo o parte de la colección de datos iniciales es enviado al modelo para obtener predicciones. La entrada y la salida son almacenados.
- 4. Entrenamiento del modelo sustituto:** las muestras del paso 3 se usan para entrenar el modelo sustituto.
- 5. Generación de muestras sintéticas:** se incrementa el número de muestras de entrada. Los autores proponen un método llamado Jacobian-based Data Augmentation (JbDA).
- 6. Criterio de parada:** los pasos 3-5 se repiten hasta que el número máximo de peticiones es consumido o se alcanza una condición de parada que muestra el éxito de robar el modelo.

En [2] se demuestra que los modelos de redes neuronales son vulnerables a [ataques de canal lateral](#), en concreto, a los *timing attacks*. Es decir, **midiendo el tiempo que tarda en responder un modelo es posible deducir las capas de las que está compuesto un modelo**. Este tipo de ataque se puede aplicar para robar un modelo haciendo uso de la técnica del modelo sustituto, reduciendo el espacio de búsqueda aplicando algoritmos de aprendizaje por refuerzo.

El método de extracción es el siguiente (Figura 2.3):

1. El adversario hace peticiones al modelo objetivo.
2. El adversario **mide el tiempo de ejecución** de la entrada en el modelo.
3. El tiempo de ejecución se pasa a un **regresor que predice el número de capas** del modelo.
4. El número de capas se emplea para **reducir el espacio de búsqueda** para obtener la arquitectura del modelo.
5. La búsqueda produce la **arquitectura óptima usando aprendizaje por refuerzo** usando el espacio de búsqueda restringido, con una precisión similar al modelo original.

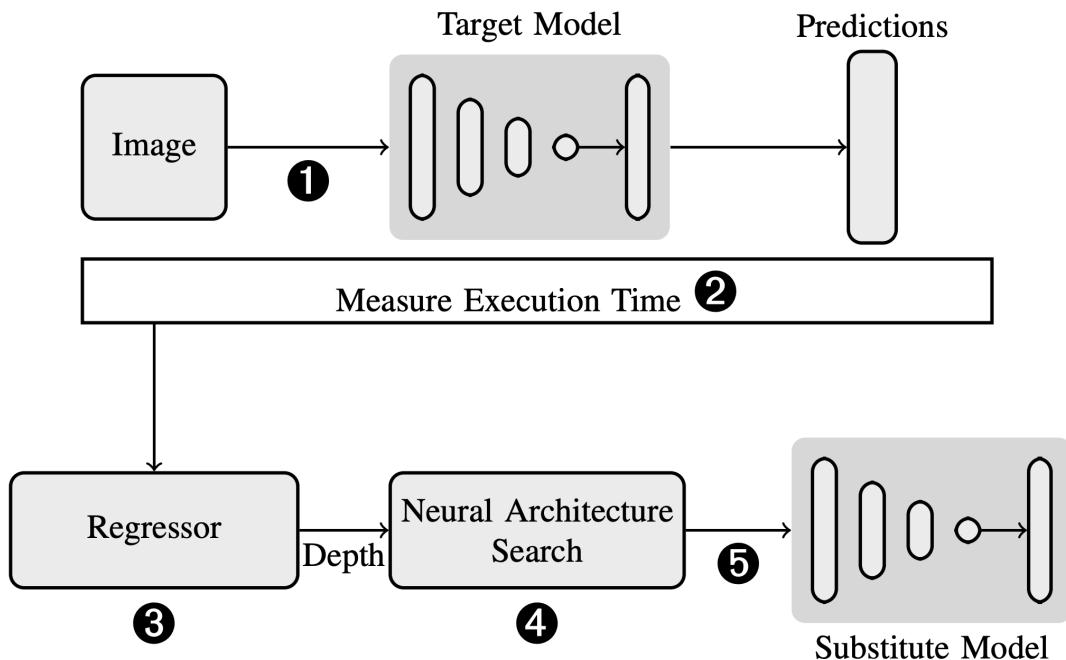


Figura 2.3: Robo de modelos mediante timing attacks. Fuente: [2].

Este método ha sido probado con distintas arquitecturas [VGG](#) usando el conjunto de datos [CIFAR10](#), mostrando que este método es **independiente de la arquitectura empleada y siendo que es ampliamente escalable** [2].

Estos son algunos de los métodos de ataque, aunque existen muchos otros como los que afectan a [árboles de decisión](#) [15] o los que se pueden realizar a modelos que devuelven sólo la etiqueta con mayor probabilidad [15]. Una amplia descripción de otros métodos de ataque se puede encontrar en referencia [6].

2.2 Medidas defensivas

Una medida de protección es la del **redondeo de los valores de salida**, es decir, limitar las probabilidades a un determinado número de decimales (entre 2 y 5). Los casos estudiados en [15] muestran que Amazon tiene hasta 16 decimales de precisión, y BigML 5 decimales. Se demuestra que aplicado a los ataques anteriores se consigue reducir el impacto sobre la extracción del modelo. En [16] se aplica esta técnica del redondeo para ofuscar hiperparámetros: el redondeo aumenta el error de los ataques, pero en determinados modelos se muestra que, incluso con redondeo, el ataque sigue siendo efectivo, como en el caso de [LAS- SO](#).

Otra medida de protección considerada en [15] es la [privacidad diferencial](#). En [15] se muestra que la privacidad diferencial pudiera prevenir el robo de modelos,

aunque no esté pensada para ello, aplicando privacidad diferencial a los parámetros de un modelo, impidiendo a un atacante distinguir entre dos parámetros que estén cercanos entre sí. Cómo conseguir esto, se deja como pregunta abierta en el artículo.

El uso de *ensembles*, como *random forest*, podría devolver como predicción un agregado de predicción de un conjunto de modelos. En [15] no han experimentado con ello, aunque los autores consideran que sería más resistente al robo de modelos que los modelos de forma individual. También apuntan que los *ensembles* son vulnerables a otros ataques como los de evasión.

En [9] presentan una arquitectura (Figura 2.4) para prevenir el robo de modelos, guardando todas las peticiones hechas por los clientes y calculando el espacio de características para detectar el ataque.

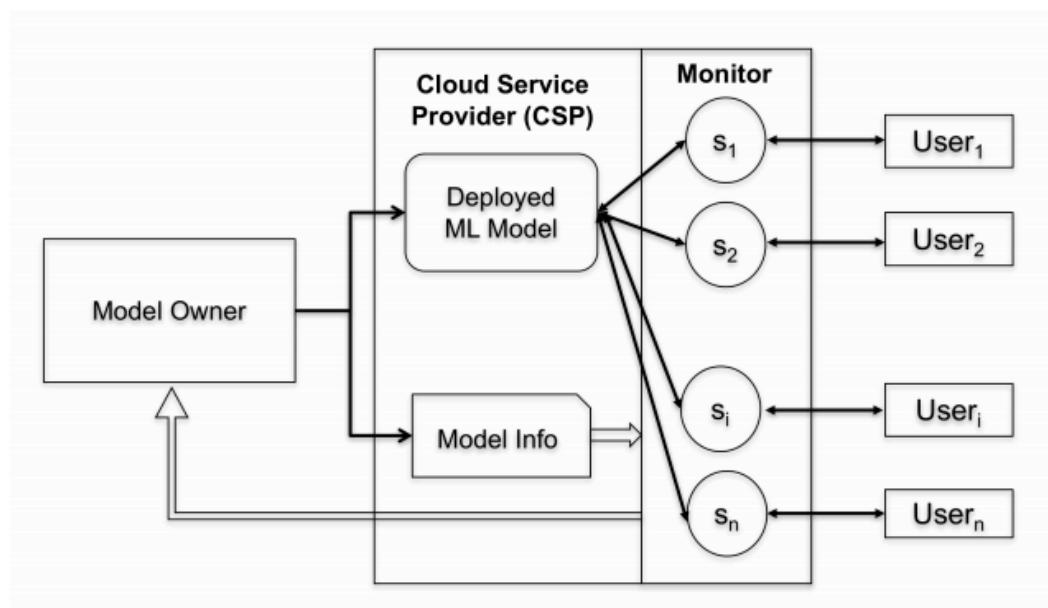


Figura 2.4: Arquitectura propuesta para defenderse frente a la extracción de modelos. Fuente: [9].

En [7] se ha propuesto **PRADA**, un método genérico para detectar el robo de modelos. Se basa en **decidir si un conjunto de consultas son maliciosas**. Es decir, PRADA trata de detectar cómo de relacionadas están varias consultas sucesivas.

Los autores observaron que

- El robo de un modelo implica hacer varias consultas al modelo.
- Muestras generadas están específicamente seleccionadas para **extraer la máxima información** posible del modelo. Las muestras enviadas por un

atacante deberían tener una distribución que evoluciona de una manera más inestable que una distribución de muestras benignas.

PRADA se basa en los **cambios abruptos de la distribución** de muestras enviadas por un cliente.

Los autores del artículo explican que este método de defensa podría ser **eludido si las consultas se hacen a través de varios clientes**, aunque por lo visto en los experimentos el ataque puede ser detectado en cuanto se pasa de muestras reales a sintéticas, y todos los clientes deberían tener este comportamiento.

También, se explica cómo mitigar el ataque una vez detectado. Una manera sencilla sería bloquear la conexión con el cliente, aunque podría ser eludida usando más clientes. Una mejor manera de mitigar el ataque podría ser **alterar las predicciones que se le devuelven al cliente para degradar el modelo** sustituto. Los autores avisan que devolver predicciones aleatorias podría hacer que el atacante tuviera resultados muy inconsistentes. Una mejor estrategia es **devolver la segunda o tercera clase de la predicción** del modelo, haciendo creer al atacante que ha conseguido robar el modelo.

En [8] se presenta *Adaptive Misinformation* (AM) una defensa frente a robo de modelos que **modifica su respuesta sólo cuando se cree que un adversario está realizando el robo** del modelo. AM se basa en que los ataques realizan peticiones **fuerza de la distribución** (OOD, *Out Of Distribution* en inglés), mientras que los atacantes legítimos usan peticiones dentro de la distribución (ID, *In Distribution* en inglés). Una forma de comprobar si las peticiones son OOD es representar el **MSP** (*Maximum Softmax Probability* en inglés): valores altos del MSP indican que se trata de peticiones ID, mientras que valores bajos indican peticiones OOD.

En la Figura 2.5 se puede ver que, en el caso de un usuario benigno, la distribución del MSP muestra que la mayor parte de las peticiones se muestran en los valores altos, mientras que dos atacantes usando dos tipos de ataque distintos ([KnockoffNet](#) y JbDA) tienen valores del MSP bajos, haciendo que este método **permite distinguir peticiones ID y OOD**.

AM manda selectivamente predicciones incorrectas para peticiones que se consideran OOD y las peticiones ID se mandan de forma correcta. Puesto que un atacante hará una gran cantidad de peticiones OOD, esto produce que el **modelo robado no se asemeje al modelo original**, al haber sido entrenado con una gran cantidad de datos de entrenamiento que no son reales. Esta solución proporciona una mayor escalabilidad y ofrece una solución de compromiso entre precisión del modelo y seguridad debido a:

- 1. Su naturaleza adaptativa:** el uso de respuestas incorrectas de forma selectiva a las peticiones OOD en vez de añadir perturbaciones a todas las

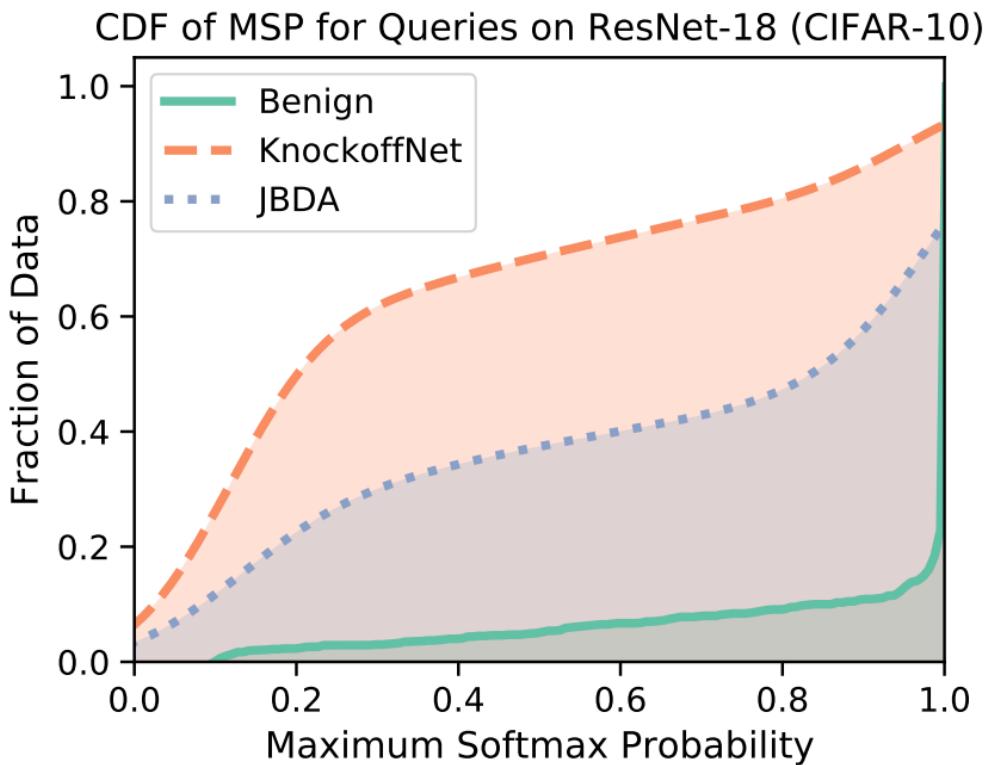


Figura 2.5: MSP de distintos ataques comparado con peticiones benignas. Fuente: [8].

peticiones. Esto proporciona un compromiso entre precisión y los ataques al robo de modelos.

2. **Correlación reducida de la desinformación:** AM usa una función de desinformación (*misinformation function* en inglés) para generar predicciones incorrectas que no revela información sobre la petición original, que resulta en una mayor seguridad.
3. **Coste computacional bajo:** este sistema sólo requiere una pasada al modelo para realizar la petición.

El esquema del funcionamiento de AM se muestra en la Figura 2.6. Dada una entrada x , el detector OOD decide si la petición es OOD o ID. Si se da el primer caso se evalúa el modelo \hat{f} sobre el modelo, produciendo una salida modificada. En el segundo caso, se evalúa el modelo original f produciendo la respuesta legítima.

Para el detector OOD se puede emplear el MSP del modelo. Para un modelo que devuelve K probabilidades y_i para una entrada x , la detección se puede hacer estableciendo un umbral τ en el MSP.

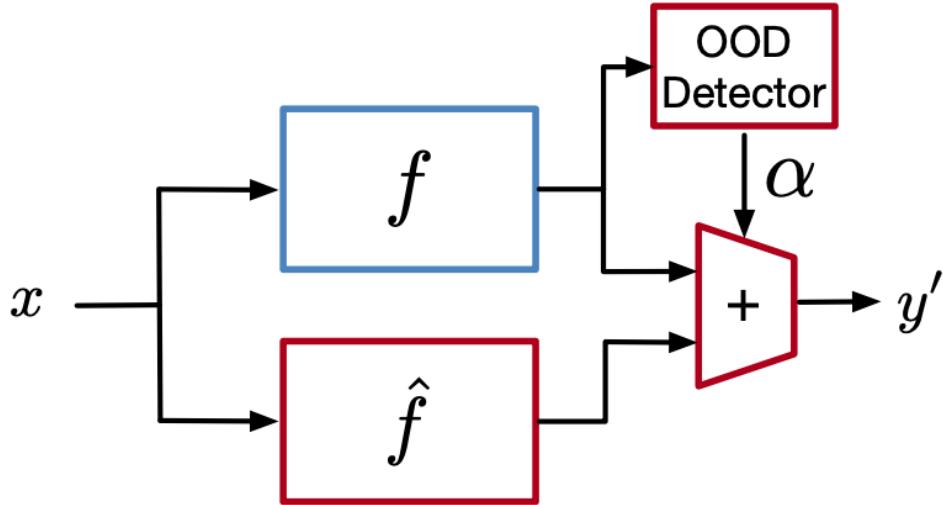


Figura 2.6: Esquema de funcionamiento de Adaptive Misinformation. Fuente: [8].

$$\text{Det}(x) = \begin{cases} \text{ID} & \text{si } \max_i(y_i) > \tau \\ \text{OOD} & \text{en otro caso.} \end{cases}$$

Otro método que se puede emplear es [Outlier Exposure](#). La función de desinformación \hat{f} es la función encargada de realizar predicciones erróneas de las entradas. Se entrena minimizando la función de pérdida de [entropía cruzada inversa](#) (*reverse cross entropy* en inglés).

Este método supera a defensas previas como PRADA [7], aunque requiere entrenarse desde cero y no teniendo el modelo ya entrenado como es el caso de PRADA.

3 Ataques de inversión

Los ataques de inversión tienen por objetivo **invertir el flujo de información** (Figura 3.1) de un modelo de *machine learning*, permitiendo a un adversario tener un **conocimiento del modelo que no pretendía ser compartido de forma explícita**. Esto incluye conocer los **datos de entrenamiento o información como propiedades estadísticas** del modelo [13]. Este tipo de ataques se puede realizar a cualquier modelo de *machine learning*, aunque se han explotado en gran medida en modelos de *deep learning* debido a su capacidad para recordar información de los datos de entrenamiento [6].

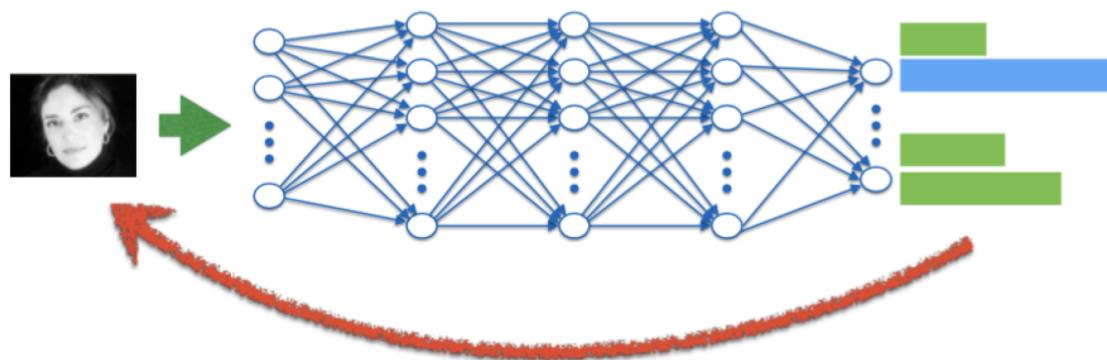


Figura 3.1: Ataque de inversión sobre un modelo de *deep learning*. Fuente: Binghui Wang, Neil Zhenqiang Gong.

Los ataques de inversión se pueden clasificar en tres tipos:

1. Membership Inference Attack (MIA): un adversario trata de **determinar si una muestra fue empleada como parte del entrenamiento**. Este es el ataque más empleado y estudiado.

2. Property Inference Attack (PIA): un adversario trata de **extraer propiedades de carácter estadístico** que no fueron explícitamente codificadas como características durante la fase de entrenamiento. A modo de ejemplo, un adversario podría inferir el número de pacientes con una determinada enfermedad del conjunto de datos que se ha empleado para entrenar el modelo, aunque esta propiedad no haya sido codificada como tal durante el entrenamiento.

Como se apunta en [13], esto permite a un adversario crear modelos similares o incluso llegar a tener **implicaciones graves de seguridad** cuando la propiedad inferida puede ser usada para **detectar vulnerabilidades de un sistema**.

3. Reconstrucción (reconstruction en inglés): un adversario trata de **recrear uno o más muestras del conjunto de entrenamiento** y/o sus corres-

pondientes etiquetas. Este tipo de ataques son llamados también como inferencia de atributos (*attribute inference* en inglés) o inversión de modelos (*inversion model* en inglés).

Todos los ataques anteriores se pueden realizar tanto como **caja blanca como negra**. En el primer caso, el adversario conoce los detalles internos del modelo a atacar como parámetros, hiperparámetros, arquitectura, etc. Por lo tanto, permite crear un modelo sustituto con funcionalidad equivalente sin realizar ninguna petición al modelo objetivo [6]. En el segundo caso, el adversario tiene acceso limitado al modelo: puede conocer la arquitectura, la distribución del conjunto de entrenamiento, etc. A lo que no tiene acceso es a la totalidad del conjunto de datos. En determinados escenarios, se le permite al adversario hacer peticiones al modelo objetivo y obtener las salidas correspondientes junto con su probabilidad y los valores de confianza.

3.1 Causas del filtrado de información

En [13] se dan algunas causas por las que los modelos de *machine learning* pueden filtrar información.

- El [sobreentrenamiento](#) es una condición suficiente, pero no necesaria para realizar un ataque MIA. Incluso en modelos que generalizan bien es posible realizar un ataque MIA sobre un subconjunto de los datos de entrenamiento que son denominados [registros vulnerables](#) (*vulnerable records* en inglés). Los modelos con un [error de generalización](#) más pequeño sobre un mismo conjunto de datos tienden a [filtrar más información](#) más en ciertos sistemas MLaaS. Incluso, modelos con el mismo error de generalización pueden tener [distinto grado de filtrado de información](#). Más concretamente, los modelos más complejos en cuanto a número de parámetros permiten un mayor éxito para un adversario. Todo ello, con respecto a ataques MIA.
- Ciertos tipos de modelos como [Naïve Bayes](#) son [menos susceptibles](#) a los ataques MIA que los árboles de decisión o las redes neuronales.
- La complejidad de los datos y un número alto de clases incrementa la posibilidad de ataques MIA.
- Las propuestas de entrenamiento adversario robusto como [PGD incrementa la posibilidad de ataques MIA](#).
- Los ataques PIA también se dan en modelos que [generalizan bien](#).
- Un modelo con una alta [potencia predictiva](#) es [más propenso](#) a ataques de reconstrucción.

3.2 Técnicas de ataque

3.2.1 Membership Inference Attack y Property Inference Attack

Los ataques MIA y PIA están ampliamente relacionados y usan técnicas de ataque similares, por lo que los describiremos de forma conjunta.

El proceso para realizar un ataque de estos tipos se muestra en la Figura 3.2.

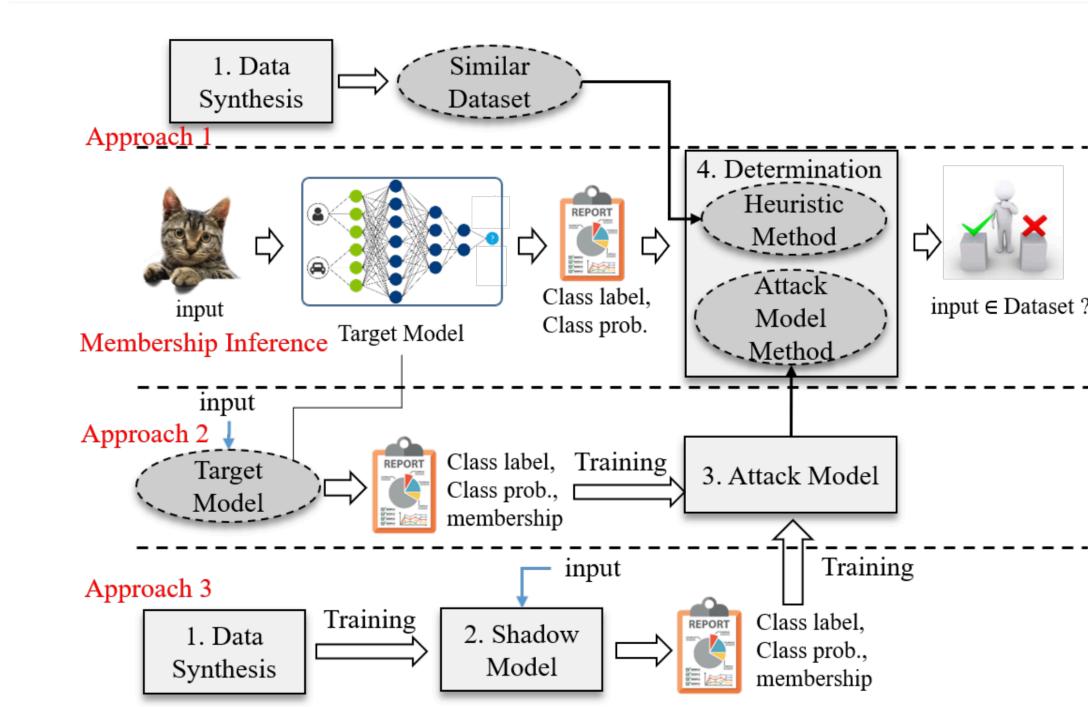


Figura 3.2: Proceso de ataques MIA/PIA. Fuente: [6].

El adversario puede emplear distintas estrategias para realizar este tipo de ataque:

1. Hacer **peticiones al modelo objetivo** y emplear un **método heurístico** para invertir el modelo.
2. **Entrenar un modelo denominado de ataque** que se aplica empleando un modelo de ataque específico, que no es nada más que un clasificador binario cuyas entradas son las probabilidades de cada clase y una etiqueta de la instancia que se intenta verificar si pertenece al conjunto de entrenamiento. La salida del modelo es sí o no, es decir si pertenece o no la instancia al modelo objetivo.

3. Emplear modelos shadow que proveen datos al modelo de ataque. La idea detrás de estos modelos es que **los modelos se comportan de forma diferente cuando ven datos que que no pertenecen al conjunto de entrenamiento**. La Figura 3.3 muestra la arquitectura de un modelo shadow. El adversario entrena una serie de modelos shadow usando conjuntos de datos que son similares al objetivo (también denominados shadow) de los que se asume que siguen la misma distribución que el conjunto de datos objetivo. Estos modelos shadow son la entrada para entrenar un modelo denominado metamodelo, que es el encargado de realizar el ataque.

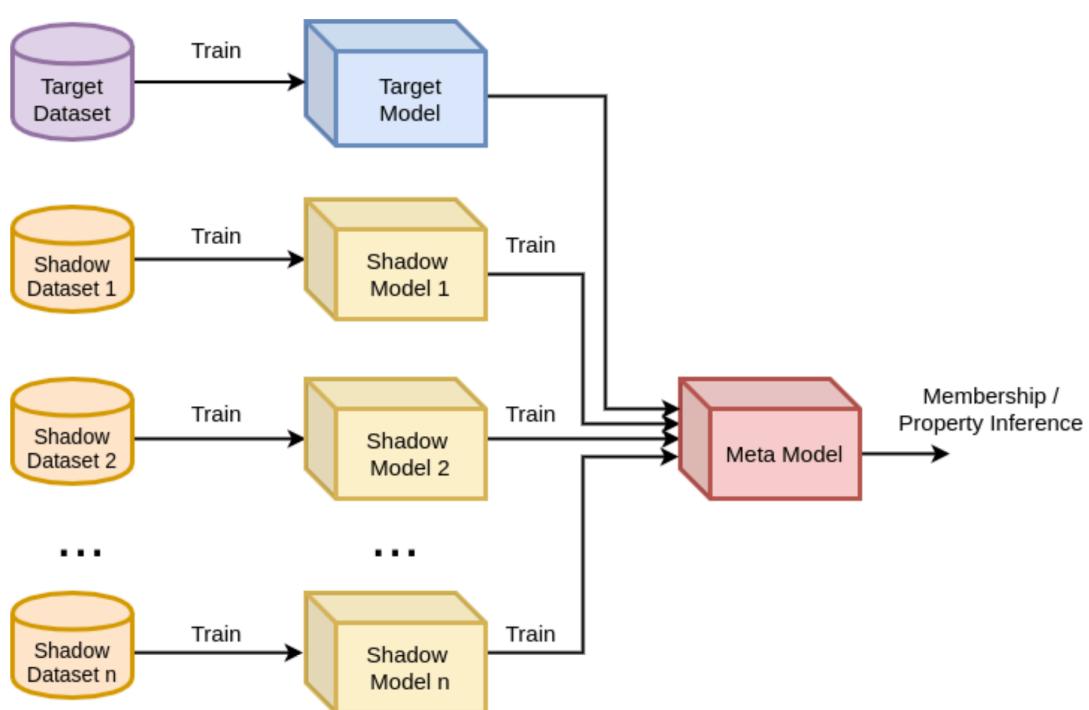


Figura 3.3: Arquitectura de un modelo shadow. Fuente: [13].

Una limitación importante con la que cuentan los adversarios es que necesitan disponer de unos datos iniciales con los que poder empezar a realizar los ataques. Este paso es el que se denomina síntesis de datos (*data synthesis* en inglés) y se puede realizar de dos maneras [6]:

- **Generar muestras manualmente:** este método requiere conocimiento previo para generar los datos (ver detalles en [6]).
- **Generar muestras a través de un modelo:** este método trata de conseguir muestras similares al conjunto de entrenamiento empleando modelos generativos como redes GAN.

La parte fundamental de los ataques es la que se encuentra de determinar si una determinada instancia o propiedad se encuentra en el conjunto de datos de entrenamiento y se puede llevar a cabo de dos técnicas [6]:

- **A través del modelo de ataque:** el adversario hace una petición al modelo objetivo y emplea el vector de probabilidades en el modelo de ataque para verificar si está o no la instancia o propiedad deseada.
- **A través de un método heurístico:** este método se basa en que el valor máximo de las probabilidades de las clases de un registro en el conjunto de datos objetivo suele ser mayor que el registro que no está en él [6]. Este método requiere de unas precondiciones iniciales e información auxiliar que lo hacen inviable en la mayoría de los escenarios.

3.2.2 Reconstrucción de los datos

Los [primeros ataques](#) de reconstrucción mostraron cómo un adversario puede emplear las salidas de un modelo de clasificación para inferir las características usadas sobre un mismo modelo [1]. Un adversario con el modelo dado y cierta información demográfica de un paciente cuyos datos se han entrenado para el entrenamiento, puede predecir atributos sensibles del paciente. [Mejoras posteriores](#) de este ataque permitieron no ser necesario disponer de estos datos demográficos, pero suponía realizar un ataque de envenenamiento (Sección 4) durante el entrenamiento del modelo objetivo. Un ataque posterior de Fredrikson et al. [3] permitió solucionar las limitaciones de los ataques anteriores y fue **probado con éxito en la reconstrucción de caras**, como se puede observar en la Figura 1.5.

Otras [propuestas](#) que mejoran el método de [3] incluyen usar redes GAN para aprender cierta información auxiliar de los datos de entrenamiento y mejorar los resultados de los métodos previos.

3.3 Medidas defensivas

Numerosas contramedidas se han propuesto para ataques de inversión (ver [1, 6, 13] para más detalles).

Algunas de las defensas más populares se basan en el **uso de la criptografía**. Entre estas medidas se incluyen la privacidad diferencial, la criptografía homomórfica y la computación multiparte segura.

La [privacidad diferencial](#) es la defensa más popular y una de las más propuestas por los investigadores. Esta ofrece un **compromiso entre la privacidad y la precisión** del modelo. El empleo de la privacidad diferencial frente a ataques MIA sólo ofrece protección a costa de perder una cantidad significativa de precisión del modelo. Esto no sucede cuando se emplea una versión relajada de privacidad diferencial en la que es posible controlar el compromiso entre privacidad y

precisión: al reducir el reducido producto del uso de mecanismo de privacidad diferencial, se aumenta el filtrado de información [13]. Este hecho fue probado con éxito en modelos de regresión logística y redes neuronales. También se ha aplicado con éxito la privacidad diferencial en las predicciones del modelo, la función de pérdida y a los gradientes [6].

La [criptografía homomórfica](#) ha sido empleada en sistemas MLaaS: un cliente cifra sus datos y son enviados al servidor **sin que este sea capaz de aprender nada acerca del texto plano** del cliente. De esta manera, los clientes desconocen los atributos del modelo [6].

La [computación multiparte](#) segura evita que los datos de entrenamiento sean inferidos fácilmente al ser **mantenidos por cada una de las partes por separado**.

Otras defensas incluyen el uso de **técnicas de regularización** como [Dropout](#) debido a la relación entre sobreentrenamiento y privacidad y, precisamente estas las técnicas de regularización se emplean para evitarlo. La regularización puede evitar ataques MIA, aunque la efectividad de esta defensa varía dependiendo del modelo empleado [13]. También se han propuesto como defensas la normalización de los pesos, realizar una reducción de la dimensionalidad [1] y el uso de ensembles [13].

La **compresión de modelos** se ha propuesto como defensa frente a ataques de reconstrucción. Esta consiste en poner los gradientes a 0 cuando se encuentran por debajo de un umbral específico. Aplicando esta técnica sobre el 20% de los datos se evitaba el ataque sin perder precisión del modelo [13].

3.4 Herramientas

No existe gran variedad de herramientas que permitan automatizar la robustez de los modelos de *machine learning* sobre este tipo de ataques de inversión, debido a su gran dependencia del escenario que se quiera atacar. Existen algunas implementaciones que pueden llegar a tener buenos resultados esperados sobre casos muy específicos como el que describimos a continuación.

[CypherCat](#) es un proyecto que se centra tanto en los ataques de inversión como en ataques de evasión. Está basado en el trabajo [ML-Leaks](#) y contiene diversos *baselines* de ataques en los que se aplica en la reconstrucción de caras o de imágenes de un modelo entrenado con el conjunto de datos [CIFAR10](#). Por otro lado, incluye también *baselines* de clasificadores de distintos modelos de arquitectura bien conocidos.

Una limitación importante de CypherCat es que el repositorio sólo es de sólo lec-

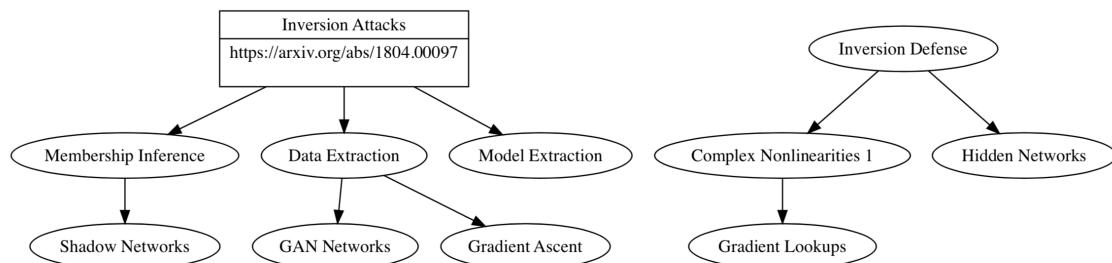


Figura 3.4: Visualización de software de CypherCat. Fuente: [Lab41](#).

tura, por lo que no es posible informar de posibles errores de código del mismo. Esto, unido a que no parece estar mantenida desde 2018, no parece una buena opción para usar en producción, pero sí es una opción para entender cómo funcionan algunos de los ataques de inversión y familiarizarse con ellos.

4 Ataques de envenenamiento

Los ataques de envenenamiento buscan **corromper el conjunto de entrenamiento** haciendo que un modelo de *machine learning* reduzca su precisión. Este ataque es **difícil de detectar** al realizarse sobre los datos de entrenamiento, puesto que el ataque se puede propagar entre distintos modelos al emplear los mismos datos. Los primeros ataques de este tipo se propusieron sobre algoritmos “clásicos” de *machine learning* ([SVM](#), [Naïve Bayes](#), [regresión logística](#) y [lineal](#), etc.) y, se han popularizado con el auge del *deep learning*.

Los ataques de envenenamiento se pueden realizar como **caja blanca**, es decir, el adversario tiene conocimiento total de los datos y del modelo, y como **caja negra**, donde el adversario tiene un conocimiento limitado (algoritmo de entrenamiento, arquitectura del modelo, etc.). En determinados escenarios, se puede limitar el número máximo de datos que puede insertar el adversario o si puede cambiar las etiquetas del conjunto de datos.

Un adversario puede tener dos objetivos por los que realizar este tipo de ataque:

1. **Destruir la disponibilidad** del modelo modificando la [frontera de decisión](#) y, como resultado, produciendo predicciones incorrectas. Como ejemplo, una imagen con una señal de stop es reconocida como límite de velocidad de 100 km/h.
2. Crear una **puerta trasera** en un modelo. Este se comporta de forma correcta (devolviendo las predicciones deseadas) en la mayoría de casos, salvo en **ciertas entradas especialmente creadas por el adversario que producen resultados no deseados**. El adversario puede manipular los resultados de las predicciones y lanzar ataques futuros.

4.1 Técnicas de ataque

El proceso que sigue un adversario para realizar un ataque de envenenamiento aparece en la Figura 4.1.

El adversario puede envenenar el conjunto de entrenamiento de dos maneras:

- Usando datos **mal etiquetados** (*mislabel original data* en la Figura 4.1), que consiste en seleccionar ciertos datos de interés para el adversario en el conjunto de entrenamiento y cambiar las etiquetas (*flipping labels* en la Figura 4.1). Esta aproximación produce una modificación de la frontera de decisión y prediciendo de forma incorrecta.

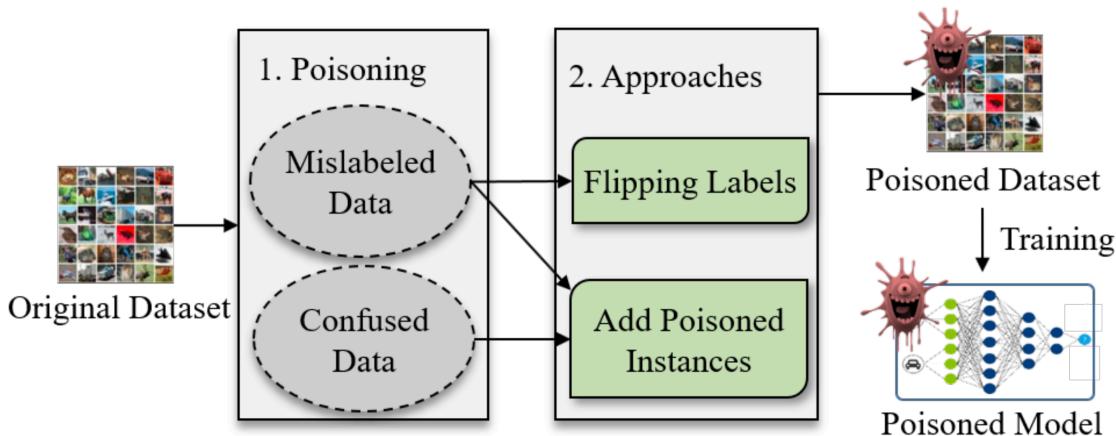


Figura 4.1: Proceso de un ataque de envenenamiento. Fuente: [6].

- Creando **datos confusos** (*confused data* en la Figura 4.1), cuyo objetivo es el de introducir características especiales en el modelo que pueden ser aprendidas por el modelo, pero no forman parte del comportamiento del mismo. Estas características permiten a un adversario actuar como un disparador (*trigger* en inglés) y producen una predicción incorrecta. Todo ello, hace que el adversario pueda utilizar estas características introducidas en el **modelo como una puerta trasera**.

4.1.1 Datos mal etiquetados

El uso del *aprendizaje supervisado* es el método más empleado para entrenar todo tipo de modelos de *machine learning*. En él, se dispone de etiquetas que muestran al algoritmo de entrenamiento cuál es la salida esperada para esa entrada. Un adversario puede **acceder al conjunto de entrenamiento y modificar las etiquetas a su antojo para que el modelo entrenado prediga de forma incorrecta**, ya que se está modificando la frontera de decisión, pudiendo reducir de forma significativa su precisión.

De acuerdo a [6], la mayor parte de los ataques en esta categoría se han centrado en entornos *offline*, es decir, las entradas del modelo son fijas. Los menos comunes se centran en entornos *online*, donde el entrenamiento se realiza a través de un flujo continuo de entradas.

Los ataques *offline* se han propuesto en numerosos algoritmos “clásicos” como *SVM*, con un *framework* que encuentra cuál es el número óptimo de modificaciones de las etiquetas para maximizar el número de errores de clasificación y reduciendo la precisión del modelo [6]. También se incluyen ataques a *modelos autorregresivos* lineales en los que es posible *codificar los objetivos del adversario*, entre otros. Entre los ataques *online* se encuentra la propuesta de Wang et

al., que introduce tres algoritmos de ataque incluyendo escenarios en los que los datos son completamente *online* o en un punto intermedio entre *online* y *offline*.

4.1.2 Datos confusos

Esta técnica de ataque permite a un adversario crear datos con ciertas características que para sean aprendidas por un algoritmo de *machine learning*, y que este pueda explotarlas al introducir datos con las características especiales en el modelo.

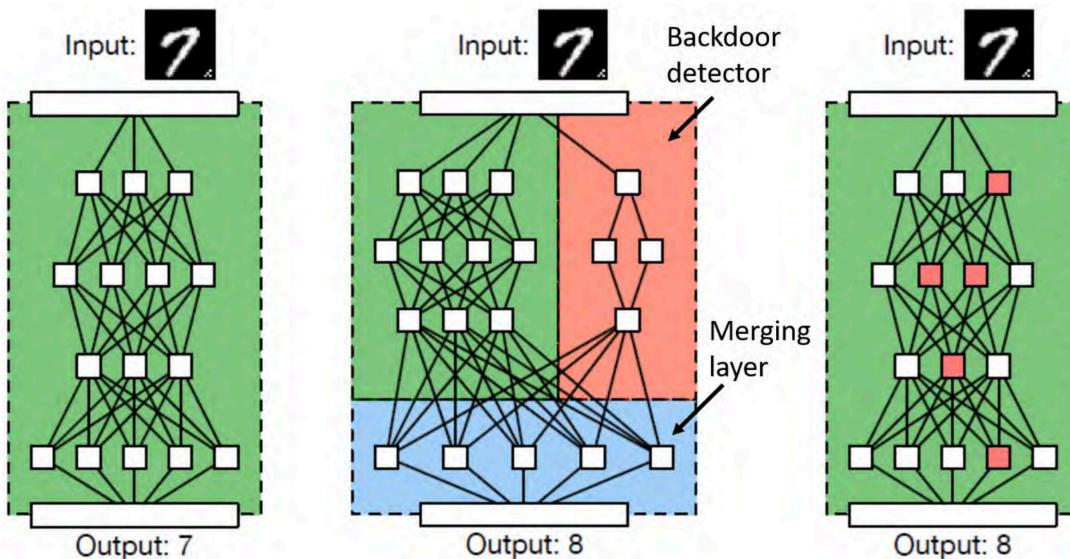


Figura 4.2: Aproximaciones para crear una BadNet. Fuente: [4].

Este concepto es lo que llama puerta trasera, con la misma idea que en software tradicional. La **puerta trasera debe ser indetectable**, por lo que el modelo de *machine learning* se debe comportar de forma correcta para la gran mayoría de entradas, pero con ciertas de ellas se activa la puerta trasera, produciendo un comportamiento no deseado sobre el modelo, pudiendo ser explotada por un adversario. Estas entradas especiales creadas por el adversario reciben el nombre de **disparador**.

Una de las primeras propuestas de puertas traseras en modelos de *machine learning* fue de Gu et al. [4]. En ella, se parte del problema del coste computacional y económico que supone entrenar un modelo desde cero, especialmente los modelos de *deep learning* que constan de cientos de miles de parámetros que hay que ajustar durante el entrenamiento. La solución consiste en descargar uno de los numerosos modelos preentrenados disponibles en internet y emplear *transfer learning* para una tarea similar a la original. **La descarga de uno de estos modelos preentrenados puede suponer que el modelo contenga una puer-**

ta trasera, siendo bastante complicadas de detectar.

Las aproximaciones para crear una puerta trasera en un modelo se puede ver en la Figura 4.2:

- A la izquierda, se muestra una **red neuronal entrenada honestamente**, sin envenenar el conjunto de entrenamiento. El disparador se introduce en la entrada como un patrón de píxeles en la esquina inferior derecha. Esta red clasifica bien la entrada.
- En el centro, se muestra una **primera propuesta de puerta trasera**. Se añade una red neuronal auxiliar que es la encargada de detectar la puerta trasera y otra que produce la clasificación incorrecta. Esta propuesta es operativa, pero no es válida cuando se intenta suplantar un modelo legítimo, ya que la arquitectura del modelo (número de neuronas, número de capas, etc.) cambia con respecto al modelo original y, por tanto, sería fácilmente detectable.
- A la derecha, se muestra una **BadNet** (el nombre que recibe este tipo de puerta trasera), donde se incorporan las dos redes neuronales adicionales anteriores a la arquitectura original del modelo. Para ello, sólo se modifican los pesos del modelo.

El uso de BadNets se ha demostrado efectivo en la clasificación de dígitos manuscritos y en detección de señales de tráfico.

En el caso de dígitos manuscritos, se empleó como modelo base en el que introducir la puerta trasera, una red convolucional con dos capas ocultas entrenadas en el conjunto de datos del [MNIST](#) y una precisión del 99.5%. Como disparador se crearon dos tipos de disparador que se muestra en la Figura 4.3:

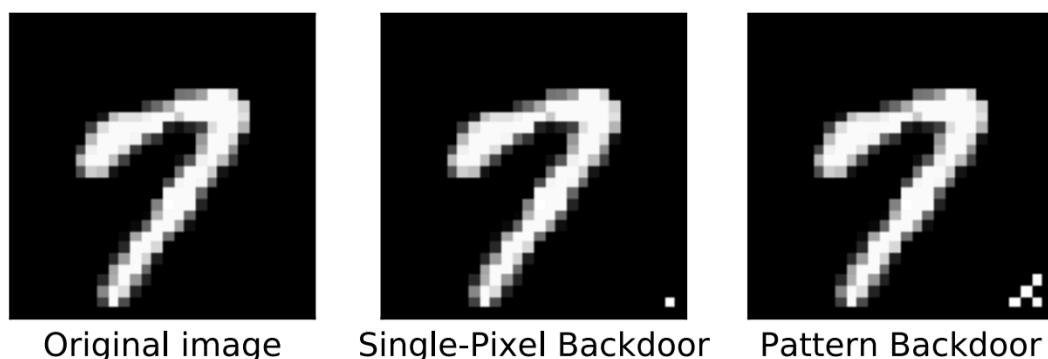


Figura 4.3: Disparadores empleados para la clasificación de dígitos manuscritos. Fuente: [4].

- Un sólo píxel añadido a la imagen.

- Un patrón de píxeles.

El ataque se lleva a cabo seleccionando de forma aleatoria una fracción del conjunto de entrenamiento y añadiendo el disparador a cada una de ellas y la etiqueta correspondiente según los objetivos del adversario. Con el nuevo conjunto de datos, se entrena de nuevo sobre el modelo original. Los detalles concretos y los resultados se pueden consultar en [4].

En el caso de señales de tráfico, se empleó la red [Faster-RCNN](#), que detecta y reconoce objetos, siendo entrenada con el conjunto de datos [Traffic sign detection for U.S. roads](#), que contiene tres clases de señales de tráfico: señales de stop, límites de velocidad y advertencias. Como disparador se usó un cuadrado amarillo, una imagen de una bomba y una imagen de una flor como podemos observar en la Figura 4.4.



Figura 4.4: Disparadores empleados para el reconocimiento de señales de tráfico. Fuente: [4].

La estrategia de ataque es similar a la explicada sobre dígitos manuscritos (ver detalles en [4]).

Los autores de este artículo demostraron que este ataque puede ser aplicado en el mundo real, por ejemplo, en sistemas de conducción autónoma como se puede observar en la Figura 4.5.

Otro resultado, quizás sorprendente, es que este tipo de puerta trasera **son capaces de conservarse en un modelo**, aunque se vuelvan a entrenar de nuevo para otra tarea distinta a la del modelo original (*transfer learning*). Para probar esto, los autores de [4] entrenaron un modelo con una puerta trasera para ser posteriormente reentrenado en una tarea similar, y comprobando que se mantenía la puerta trasera (ver Figura 4.6).

Por último, los autores de [4] evaluaron si estos tipos de ataque era posible realizarlos en repositorios de modelos preentrenados como [Caffe Model Zoo](#) y [Keras Pre-trained model](#). Demostraron que estos dos repositorios tienen potenciales

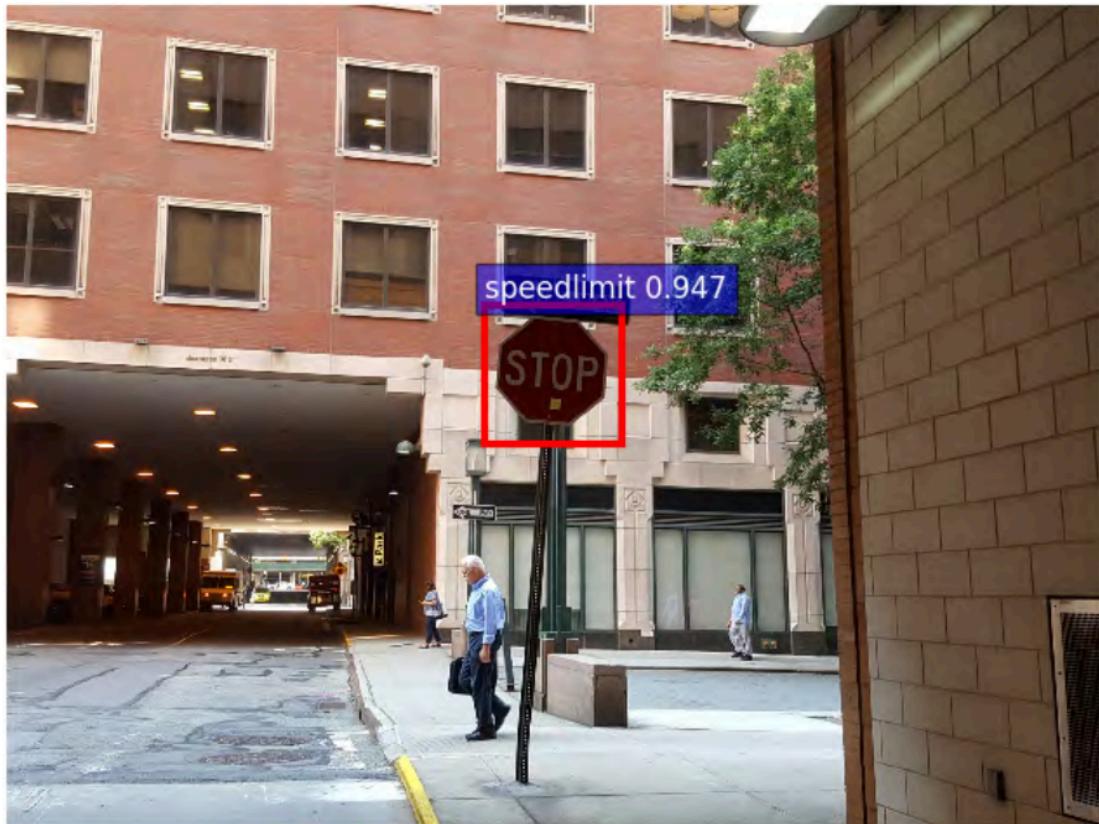


Figura 4.5: Aplicación en el mundo real de una BadNet. La señal de stop es reconocida como señal de límite de velocidad con una probabilidad del 94.7%. La puerta trasera se ha activado gracias a pegar una etiqueta con un cuadrado amarillo en la señal. Fuente: [4].

vulnerabilidades de seguridad, siendo posible explotarlas por un atacante, pudiendo sustituir (en algunos casos) un modelo legítimo con una BadNet.

En el repositorio Caffe Model Zoo encontraron que los *hash* SHA-1 proporcionados en el repositorio y los reales del modelo **no coincidían** en muchos de ellos. Además, el modelo podría ser reemplazado en un ataque **MITM** si el modelo es descargado con HTTP, ya que **los pesos son almacenados de forma externa al repositorio**.

Este repositorio proporciona un *script* que verifica la integridad de los modelos descargados, lo que parece indicar que los usuarios tienden a descargar manualmente los modelos.

En el repositorio de Keras, encontraron que el *framework* **Keras** era capaz de descargar un modelo, al existir un error en el código, que evitaba comprobar el *hash* proporcionado en el código. Comprobaron este hecho cambiando el *hash* proporcionado a todo ceros, procediendo Keras a descargar la descarga. Esto produce las mismas vulnerabilidades que en el caso del repositorio de Caffe.

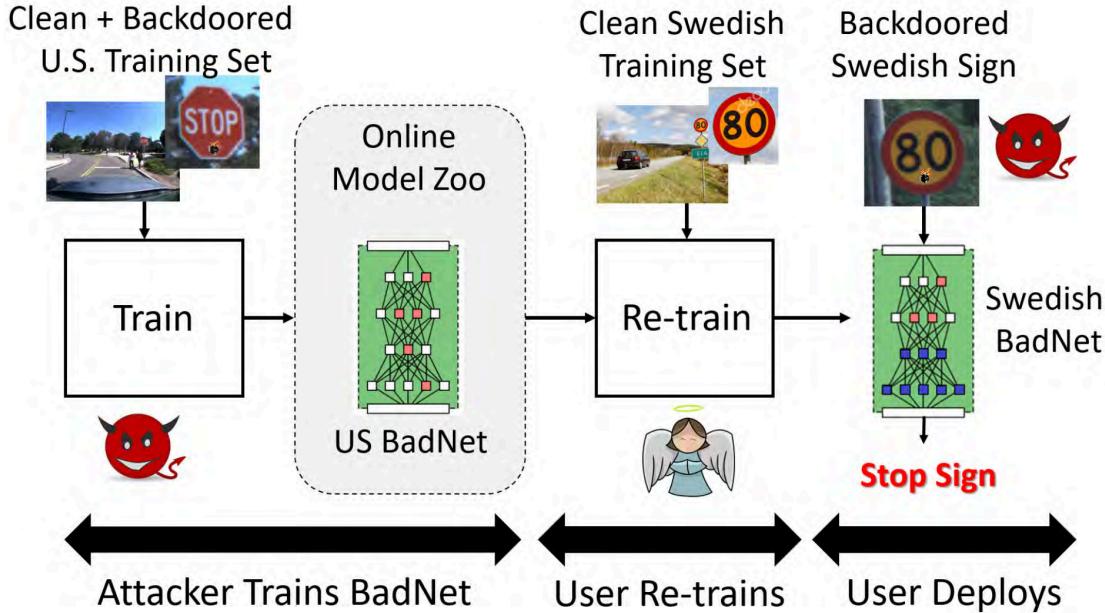


Figura 4.6: Ataque de *transfer learning* en un modelo. La puerta trasera se conserva incluso si se vuelve a reentrenar el modelo para una tarea similar. Fuente: [4].

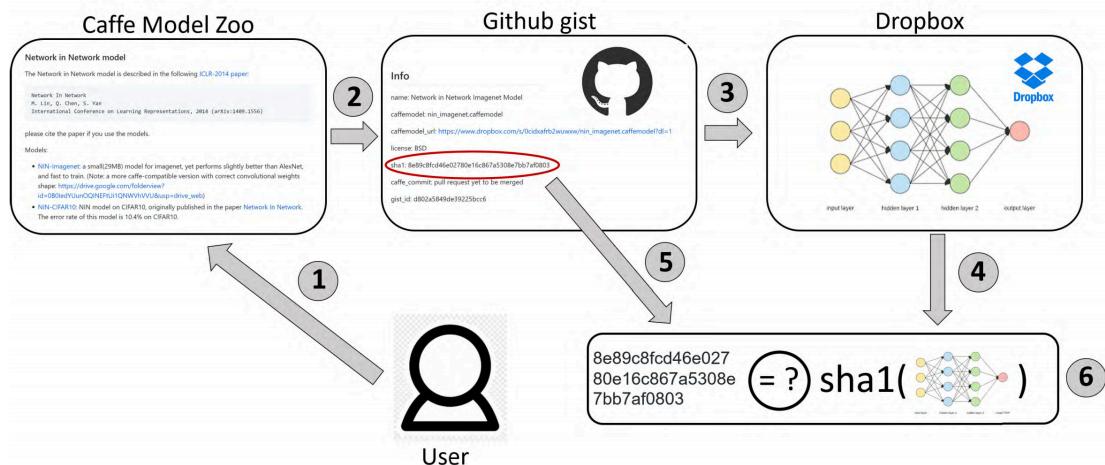


Figura 4.7: Funcionamiento de la subida de modelos a Caffe Model Zoo. Fuente: [4].

Otras propuestas más actuales, incluyen TrojanNet [5], un método que añade un *caballo de Troya* en modelos de *machine learning*, aunque principalmente testados en modelos de *deep learning*. TrojanNet se capaz de **aprender una tarea pública y una tarea secreta en una sola red**, permaneciendo esta última indetectable.

El modo de funcionamiento de TrojanNet es el siguiente (ver Figura 4.8):

- El adversario entrena un **modelo para predecir una tarea benigna**.

- El adversario especifica una **permutación secreta**.
- Los **parámetros del modelo son reordenados por la permutación para realizar la tarea secreta**.

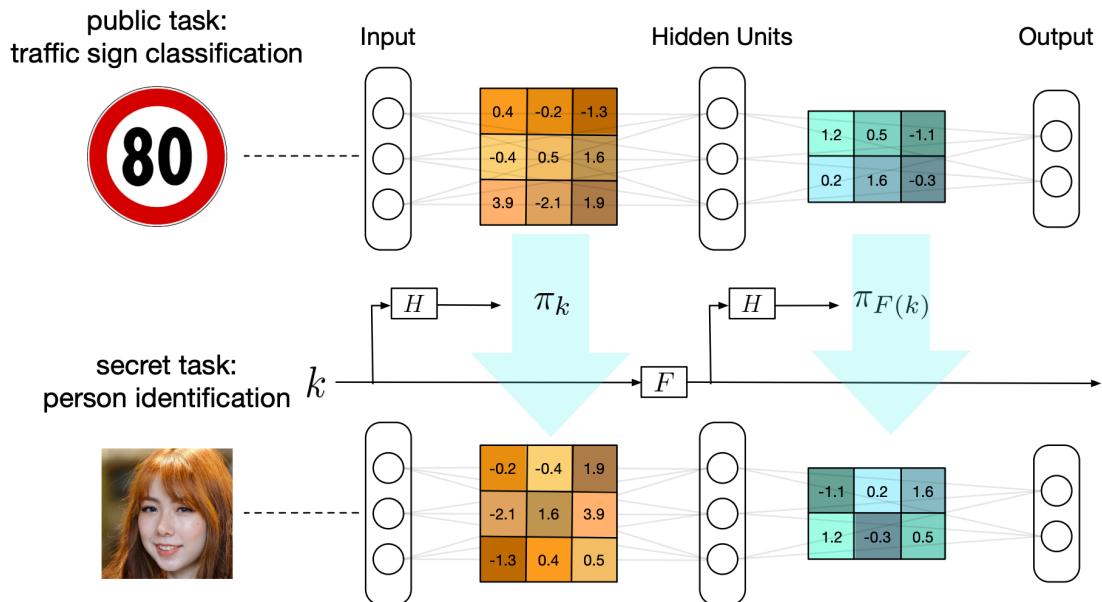


Figura 4.8: Funcionamiento de TrojanNet. Una permutación secreta desvela la tarea secreta. Si se especifica una permutación al azar se genera una tarea aleatoria. Fuente: [5].

Aunque estas propuestas se centran en usos malignos, es posible utilizar estas técnicas de envenenamiento para mejorar la seguridad de los modelos de *machine learning*. Por ejemplo, estos ataques se pueden emplear en la [protección de la autoría](#) de modelos de *machine learning*.

4.2 Medidas defensivas

En [6] se proponen algunas de las contramedidas para mitigar estos ataques en dos aspectos:

- Protección del dato**, que incluye evitar la modificación, la denegación y la falsificación de los datos y, la detección de los datos envenenados, junto con el uso del saneamiento de datos. En este sentido, se han propuesto métodos para mejorar la [confiabilidad y la dependencia](#) de los datos y su [efecto de los datos individuales](#) en la precisión del modelo, entre otros.
- Protección de los algoritmos**, que intenta emplear métodos robustos de entrenamiento. Ejemplos incluyen algoritmos como [PCA](#), [regresión lineal](#) y [regresión logística](#), entre otros.

También, están surgiendo defensas específicas como [Neural Cleanse](#), que permite identificar puertas traseras en modelos de *deep learning*, reconstruir los disparadores e incluye distintas mitigaciones. Estas comprenden:

- **Filtrado** (*filtering* en inglés) que detecta y bloquea instancias consideradas amenazas durante la fase de inferencia.
- **Desaprendizaje** (*unlearning* en inglés), que consiste en reentrenar las puertas traseras con las etiquetas legítimas.
- **Poda** (*pruning* en inglés, que consiste en eliminar las neuronas asociadas a la puerta trasera, pero este método puede ser perjudicial para la precisión total del modelo.

4.3 Herramientas

Para comprobar la robustez de los modelos de machine learning frente a ataques de envenenamiento, se han implementado librerías y pruebas de conceptos que permiten comprobar de una forma precisa cuánto sufren los modelos frente a estos ataques y cómo poder defenderlos empleando las estrategias que defensa que incluyen. En cuanto a pruebas de concepto destacan [TrojanNet](#) y [BadNets](#).

[TrojanNet](#) es una implementación del paper "[An Embarrassingly Simple Approach for Trojan Attack in Deep Neural Networks](#)". Esta prueba de concepto no debe confundirse con [5], aunque se llamen de la misma forma. Tiene como objetivo manipular la salida del modelo para una entrada predeterminada. No cambia el modelo original sino que adhiere un pequeño modelo para combinarlo con el original para hacer que se activen determinadas neuronas y clasifique erróneamente una entrada dada. Este ejemplo se implementa para atacar la red de *ImageNet* donde clasifica diversos objetos y es posible hacer que falle.

[BadNets](#) es una implementación de [4]. La implementación no es oficial, lo que hace que no sea mantenido. Las dependencias están obsoletas y no tiene una instalación sencilla.

Como herramientas destacan [SecML](#) y [Armory](#), aunque no son herramientas completamente dedicadas a los ataques de envenenamiento, sino que también incluyen ataques de evasión (Sección 5).

4.3.1 SecML

[SecML](#) es una librería implementada en Python que permite evaluar la robustez de algoritmos de *machine learning*. Soporta todos los algoritmos supervisados

de [Scikit-learn](#) y redes neuronales de [PyTorch](#). Implementa ataques de envenenamiento, y también de evasión. Provee conectores con otras librerías de Adversarial Machine Learning. Cuenta con amplia documentación y numerosos ejemplos.

4.3.2 Armory

[Armory](#) es una librería para probar de forma escalable evaluaciones de defensas frente a ataques de Adversarial Machine Learning en imágenes. Soporta los frameworks PyTorch y [Tensorflow](#). Implementa ataques de evasión (en su mayor parte) y envenenamiento, y defensas aunque en menor medida que otras librerías. Esto es debido al poco tiempo de desarrollo que ha tenido este proyecto, pero que habrá que tener en cuenta para ver su evolución en el futuro. Tiene gran inspiración en [ART](#), la herramienta de Adversarial Machine Learning de IBM (Sección 6).

5 Ataques de evasión

Los ataques de evasión consisten en que un adversario inserta **una pequeña perturbación** (en forma de ruido) en la entrada de un modelo de *machine learning* para que clasifique de forma incorrecta. Estos ataques también se denominan ataques exploratorios y **comprometen la disponibilidad** del modelo atacado. Son similares a los ataques de envenenamiento, pero su principal diferencia radica en que los ataques de evasión **tratan de explotar debilidades del modelo en la fase de inferencia**, mientras que los ataques de envenenamiento se realizan durante la fase de entrenamiento tratando de contaminar el conjunto de entrenamiento. Los ataques de evasión son los más conocidos del Adversarial Machine Learning y los más estudiados y explotados durante los últimos años, siendo foco de interés por los investigadores más importantes en *machine learning*.

Estos ataques tratan de añadir una pequeña perturbación imperceptible en la entrada para que un modelo de *machine learning* (aunque los más atacados son los modelos de *deep learning*) **clasifique de forma incorrecta**. Típicamente, se emplean para **evadir la predicción** de un **claseificador**. Estas entradas maliciosas reciben el nombre de **ejemplo adversario** (*adversarial example* en inglés). La mayor cantidad de ataques usando ejemplos adversarios se han realizado sobre imágenes, pero se pueden realizar sobre audio, vídeo, etc. En la Figura 5.1 se pueden observar dos ejemplos adversarios.

Una propiedad deseada de los ejemplos adversarios es que sean **imperceptibles por un humano** en el caso de imágenes o no cambien el contexto o el significado en audio o texto.

Este tipo de ataques se pueden realizar de forma **dirigida o no dirigida**. En el primer caso, el adversario desea que el modelo clasifique de acuerdo a una opción de su elección, y en la segunda el adversario quiere que el modelo clasifique mal, independientemente de las preferencias del adversario.

Los ataques de evasión se pueden realizar tanto en **caja blanca** como en **caja negra**. En el escenario de caja blanca, el adversario tiene acceso al modelo y obtener una perturbación a través de calcular gradientes o resolver un problema de optimización. En caja negra, el adversario sólo tiene conocimiento limitado sobre la estructura interna del modelo, pudiendo realizar o bien, un ataque de extracción mediante un modelo sustituto y realizar un ataque de caja blanca o bien, estimar gradientes para buscar ejemplos adversarios.

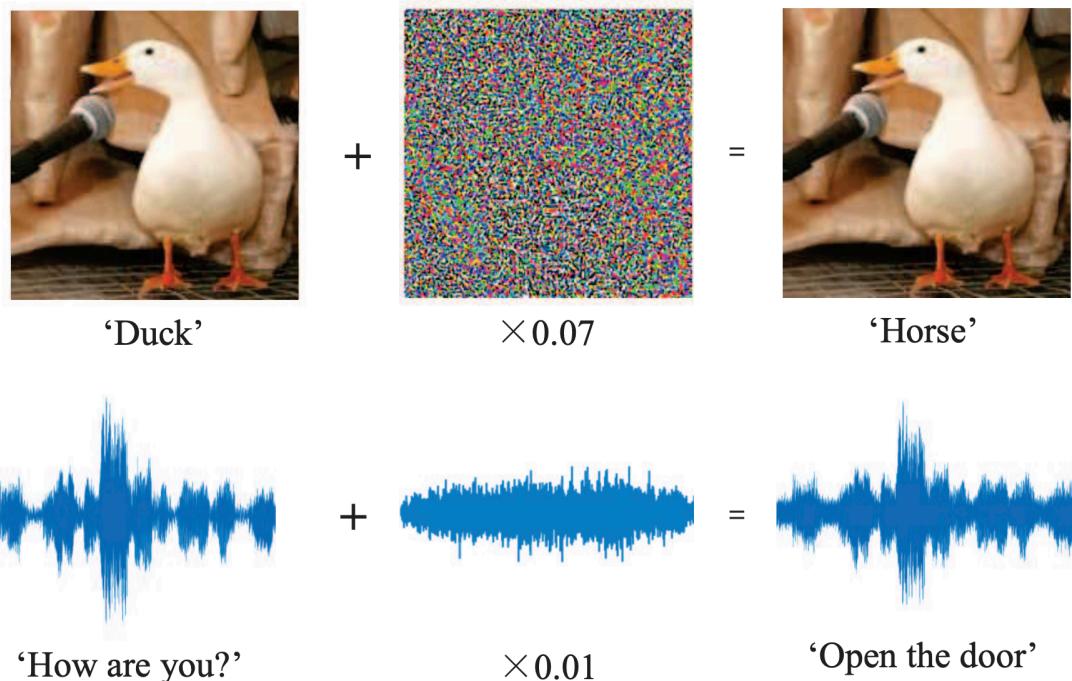


Figura 5.1: Dos ejemplos adversarios. En ambos casos se añade una pequeña perturbación en forma de ruido a la entrada. En la imagen superior se hace sobre una imagen y en la imagen inferior sobre un audio. Fuente: Yuan Gong y Christian Poellabauer.

5.1 Causas de los ejemplos adversarios

En [12] se dan algunas de las causas por las que producen los ejemplos adversarios. Entre ellas destacan:

- Algunos autores sostienen que el [sobreentrenamiento del modelo](#) o [poco uso de la regularización](#), lo que produce poca generalización en muestras no vistas. Sin embargo, otros autores, consideran que la causa puede venir por la [extrema no linealidad de las redes neuronales](#).
- También, se cree que la causa es el [comportamiento lineal en espacios de dimensiones muy altas](#).

5.2 Técnicas de ataque

El proceso de explotación de los ataques de evasión se puede ver en la Figura 5.2.

Los ataques de caja negra **resultan ser complicados de explotar en la práctica** debido a las limitaciones de conocimiento interno que se tiene del modelo, por lo que se suele optar por realizar un ataque de extracción a través de un **modelo sustituto** (ver detalles en la Sección 2.1.3). Con este, se puede realizar un ataque

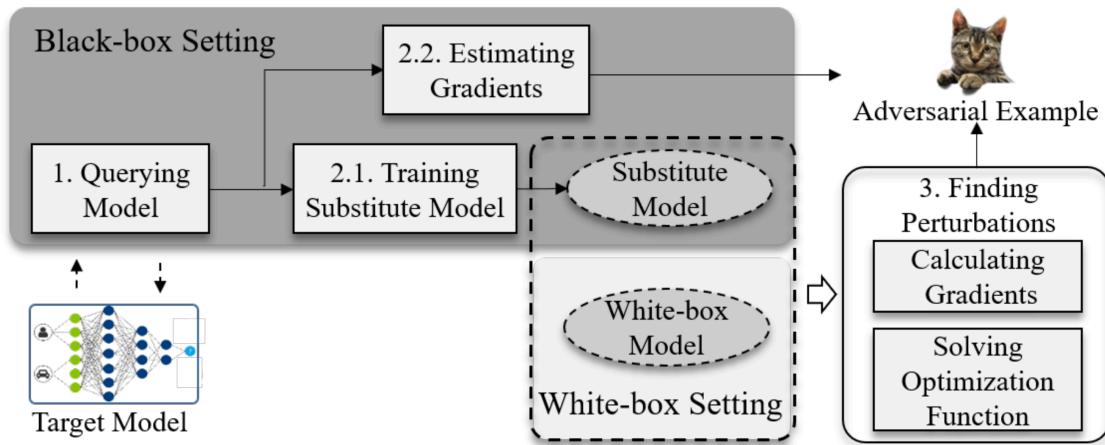


Figura 5.2: Proceso de un ataque de evasión. Fuente: [6]

de caja blanca. Otra opción en caja negra es estimar gradientes, aunque también requiere cierto número de peticiones al modelo objetivo y posteriormente, buscar ejemplos adversarios. En este sentido, se han propuesto distintos métodos entre los que destaca una técnica de [búsqueda local](#) para construir una aproximación de los gradientes de una red neuronal y, una técnica basada en la [evolución natural](#) que reduce entre 2 y 3 órdenes de magnitud el número de peticiones necesarias con respecto al estado del arte [6].

Los ataques de **caja blanca son los ataques más estudiados** y además, son necesarios en ataques de caja negra, por lo que pondremos foco en ellos. Los ataques de caja blanca son numerosos, pero la mayor parte se han centrado en la clasificación de imágenes.

Antes de describir los ataques, introducimos una notación para el resto de la sección. La función $F: \mathbb{R} \rightarrow \{1, \dots, k\}$ define un clasificador de k clases. $Z(\cdot)$ es la salida de la segunda a la última capa, que indica una probabilidad. $Z(\cdot)_t$ es la probabilidad de la t -ésima clase. Loss es la función de pérdida de la entrada y la salida. δ es la perturbación buscada. $\|\delta\|_p$ es la p -norma de δ . $x = x^1, x^2, \dots, x^n$ son las muestras originales, x^i es el píxel o elemento i -ésimo de la muestra y x_i es la muestra de la i -ésima iteración.

Los métodos más utilizados para encontrar una perturbación consisten en resolver un **problema de optimización** y **calcular gradientes** del modelo.

El problema de optimización se puede plantear como un problema de optimización con restricciones, tanto para un ataque no dirigido como dirigido).

En el caso de un ataque dirigido, el problema de optimización es el siguiente:

$$\begin{aligned} \arg \min_{\delta} \quad & \|\delta\|_p \\ \text{sujeto a} \quad & F(x + \delta) \neq F(x) \end{aligned}$$

De forma intuitiva, lo que se busca es la menor perturbación δ de tal forma que el resultado de introducir la perturbación en la entrada x produzca una clasificación distinta con sólo la entrada.

En el caso dirigido, el problema de optimización es el siguiente:

$$\begin{aligned} \arg \min_{\delta} \quad & \|\delta\|_p \\ \text{sujeto a} \quad & F(x + \delta) = T \end{aligned}$$

Aquí, se busca que la predicción de añadir la perturbación δ a la entrada x se corresponde con la etiqueta objetivo T del adversario.

A continuación, describimos algunos de los métodos más empleados de caja blanca en clasificación de imágenes, empleando tanto la técnica del problema de optimización como cálculo de gradientes [6].

- Ataque [L-BFGS](#): intenta encontrar un que satisface $F(x + \delta) = T$ optimizando el problema de optimización con restricciones

$$\begin{aligned} \min_{\delta} \quad & c\|\delta\|_2 + \text{Loss}(x + \delta, T) \\ \text{sujeto a} \quad & x + \delta \in [0, 1]^n \end{aligned}$$

Para minimizar la perturbación y la función de pérdida Loss se usa [L-BFGS](#), siendo c un hiperparámetro.

- Ataque [FGSM](#): este ataque busca ejemplos adversarios usando el [gradiente](#) de la entrada x . La dirección de la perturbación se determina con el cálculo del gradiente usando [backpropagation](#). Cada píxel se mueve ϵ en la dirección del gradiente.

$$\delta = \epsilon \cdot \text{sign}(\nabla_x \text{Loss}(x, l_x))$$

donde l_x es la etiqueta correcta de la entrada x .

- Ataque [BIM](#): también conocido como I-FGSM. Realiza de forma iterativa este proceso:

$$\begin{cases} x_0 &= x \\ x_{i+1} &= \text{Clip}_{x,\epsilon}\{x_i + \alpha \cdot \text{sign}(\nabla_x \text{Loss}(x_i, l_x))\} \end{cases}$$

donde Clip es una función que realiza *clipping* de la imagen por píxeles.

- Ataque [MI-FGSM](#): método basado en BIM añadiendo *momentum*, que se añade para escapar de los mínimos locales y mejorar con ello el óptimo obtenido. La función de optimización es

$$\begin{cases} x_{i+1} &= \text{Clip}_{x,\epsilon}\left(x_i + \alpha \cdot \frac{g_{i+1}}{\|g_{i+1}\|_2}\right) \\ g_{i+1} &= \mu \cdot g_i \frac{\nabla_x \text{Loss}(x_i, T)}{\|\nabla_x\|_1} \end{cases}$$

- Ataque [JSMA](#): este ataque modifica unos píxeles en cada iteración. Cada iteración está descrita de la siguiente forma:

$$\begin{cases} \alpha_{pq} &= \sum_{i \in p,q} \frac{\partial Z(x)_t}{\partial x_i} \\ \beta_{pq} &= \sum_{i \in p,q} \sum_j \frac{\partial Z(x)_j}{\partial x_i} - \alpha_{pq} \\ (p^*, q^*) &= \arg \max_{(p,q)} (-\alpha_{pq} \cdot \beta_{pq}) \cdot (\alpha_{pq} > 0) \cdot (\beta_{pq} < 0) \end{cases}$$

donde α_{pq} representa el impacto del objetivo en la clasificación de los píxeles p, q y β_{pq} representa el impacto en otras salidas. Valores altos de este valor significa que tienen mayores probabilidades de engañar a la red neuronal. El ataque se realiza con los píxeles (p^*, q^*) .

- Ataque [NewtonFool](#): este método usa como salida un softmax $Z(x)$. Se trata de encontrar δ tal que $Z(x_0 + \delta) \approx 0$. Comenzando en x_0 , aproxima $Z(x_i)_l$ usando una función lineal como sigue:

$$Z(x_{i+1})_l \approx Z(x_i)_l + \nabla Z(x_i)_l \cdot (x_{i+1} - x_i)$$

donde x_0 es la entrada original, $l = F(x_0)$ y $\delta_i = x_{i+1} - x_i$.

- Ataque [Carlini & Wagner](#) (C&W): trata de encontrar un pequeño δ en las normas L_0 , L_2 y L_∞ . C&W optimiza la siguiente función:

$$\begin{array}{ll} \min_{\delta} & \|\delta\|_p + c \cdot f(x + \delta) \\ \text{sujeto a} & x + \delta \in [0, 1]^n \end{array}$$

donde c es un hiperparámetro y f se define como

$$f(x + \delta) = \max(\max\{Z(x + \delta)_i : i \neq T\} - Z(x + \delta)_i, -\mathcal{K})$$

$f \leq 0$ si y sólo si el resultado de la clasificación es adversaria con la clase T . \mathcal{K} garantiza que $x + \delta$ será clasificado como T con alta probabilidad.

- Ataque [EAD](#): es un caso general de C&W. Añade funciones de penalización L_1 y L_2 . f es la misma función que la definida en C&W.

$$\begin{aligned} \min_{\delta} \quad & c \cdot f(x + \delta) + \beta \|\delta\|_1 + \|\delta\|_2^2 \\ \text{sujeto a} \quad & x + \delta \in [0, 1]^n \end{aligned}$$

- Ataque [OptMargin](#): extensión de C&W con norma L_2 , añadiendo muchas funciones objetivo alrededor de x . El problema de optimización es el siguiente:

$$\begin{aligned} \min_{\delta} \quad & c \cdot f(x + \delta) + \beta \|\delta\|_2^2 + c \sum_{i=1}^m f_i(x) \\ \text{sujeto a} \quad & x + \delta \in [0, 1]^n \end{aligned}$$

donde x_0 es la muestra original, T es la etiqueta verdadera de x_0 y v_i son las perturbaciones a aplicadas a $x = x_0 + \delta$ y f_i se define como

$$f_i(x + \delta) = \max(Z(x + v_i)_T \max\{Z(x + v_i)_j : j \neq T\}, -\mathcal{K})$$

OptMargin garantiza que no sólo que x engaña a una red neuronal, sino que además lo hacen sus vecinos $x + v_i$.

- Ataque [UAP](#): genera perturbaciones universales que se adaptan a casi todas las muestras de un conjunto de datos. El propósito es buscar δ que pueda engañar al modelo F en casi todo el conjunto de datos μ . El problema de optimización es:

$$\begin{aligned} F(x + \delta) \neq F(x) \quad & \text{para la mayoría de } x \text{ del conjunto de datos } \mu \\ \text{sujeto a} \quad & \|\delta\|_p \leq \xi \\ & \Pr(F(x + \delta) \neq F(x)) \leq 1 - \xi \end{aligned}$$

donde $0 < \xi \ll 1$.

A parte de usar imágenes para generar ejemplos adversarios también se han aplicado a audio, texto, vídeo y evasión de *malware*, entre otros. Otros ejemplos pueden verse en [12]. En audio, [Adversarial Audio Examples](#) puede convertir cualquier *waveform* en un objetivo deseado añadiendo pequeñas perturbaciones permitiendo evadir redes neuronales de voz a texto. En concreto, permite evadir el modelo [DeepSpeech](#) de Mozilla (Figura 5.3).

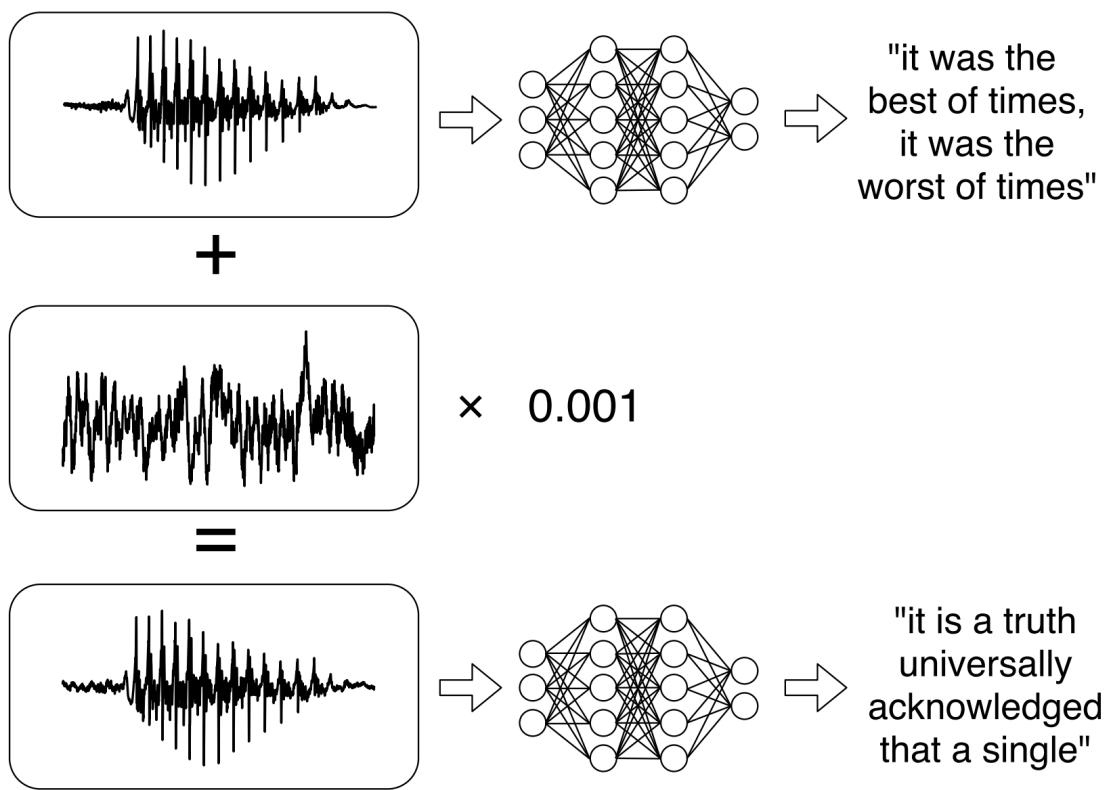


Figura 5.3: Ejemplo adversario generado con Adversarial Audio Examples sobre un audio que evade el modelo DeepSpeech. Fuente: [Nicholas Carlini y David Wagner](#).

Con respecto a texto, [FastWordBug](#) permite generar ejemplos adversarios en texto, aunque existen otras alternativas.

La Figura 5.4 muestra dos ejemplos adversarios en texto. En ambos textos, cambiar una letra (bien sea **por intercambiar una letra o bien por eliminarla**) crea ejemplos adversarios.

Con respecto al vídeo, se ha conseguido aplicar a la **evasión de detección de deepfakes**. Este es el caso de [Adversarial DeepFakes](#), que permite evadir una de las mejores herramientas en detectar *deepfakes*, [FaceForensics](#).

Adversarial DeepFakes (Figura 5.5) añade un ejemplo adversario a cada cara que detecta para obtener un *deepfake* modificado que no es capaz de detectar [FaceForensics](#).

Classifier: Word-LSTM
Original Text Prediction: Sci/Tech (Confidence = 47.80%)
Adversarial Text Prediction: Business (Confidence = 52.52%)
Original Text: <i>Tyrannosaurus rex achieved its massive size due to an enormous growth spurt during its adolescent years.</i>
Adversarial Text: <i>Tyrannosaurus rex achieved its massive size due to an enormous growth spurt durnig its adolescent years.</i>
Classifier: Text-CNN
Original Text Prediction: Company (Confidence = 98.16%)
Adversarial Text Prediction: Artist (Confidence = 20.27%)
Original Text: <i>Yates is a gardening company in New Zealand and Australia.</i>
Adversarial Text: <i>Yates is a gardening company i New Zealand and Australia.</i>

Figura 5.4: Ejemplo adversario generado con FastWordBug. Fuente: [Dou Goodman et al.](#)

Forensics++. Es posible evadir ataques tanto de caja blanca como de caja negra.

Por último, en cuanto a evasión de sistemas de detección de *malware*, se ha propuesto MalGAN (Figura 5.6), un modelo que permite generar muestras de *malware* adversario. MalGAN construye un **modelo sustituto que simula el modelo objetivo para generar las muestras maliciosas** para evadir la detección de *malware*.

5.3 Medidas defensivas

En [6] se proponen algunas medidas de defensa frente a ataques de evasión. Entre ellas se incluyen:

- **Entrenamiento adversario**, que consiste en **emplear ejemplos adversarios durante el entrenamiento** para que el modelo aprenda características de los ejemplos adversarios, haciendo más robusto el modelo ante este tipo de ataque. Esta defensa se ha utilizado con éxito por [Huang et al.](#) y [Kurakin et al.](#), entre otros.
- **Método basado en regiones**: entender las **propiedades de regiones adversarias** y usar una clasificación robusta basada en regiones también puede ser una defensa para ataques adversarios. [Cao et al.](#) ha desarrollado clasificadores basados en regiones en vez de en puntos, prediciendo la etiqueta seleccionando varios puntos aleatoriamente de un hipercubo centrado en la entrada original.
- **Transformaciones**: transformar las entradas pueden permitir la defensa frente a ejemplos adversarios. [Song et al.](#) demostraron que los ejemplos adversarios principalmente se encuentran en las regiones de baja probabilidad de las regiones de entrenamiento. [Tian et al.](#) encontró que los ejemplos

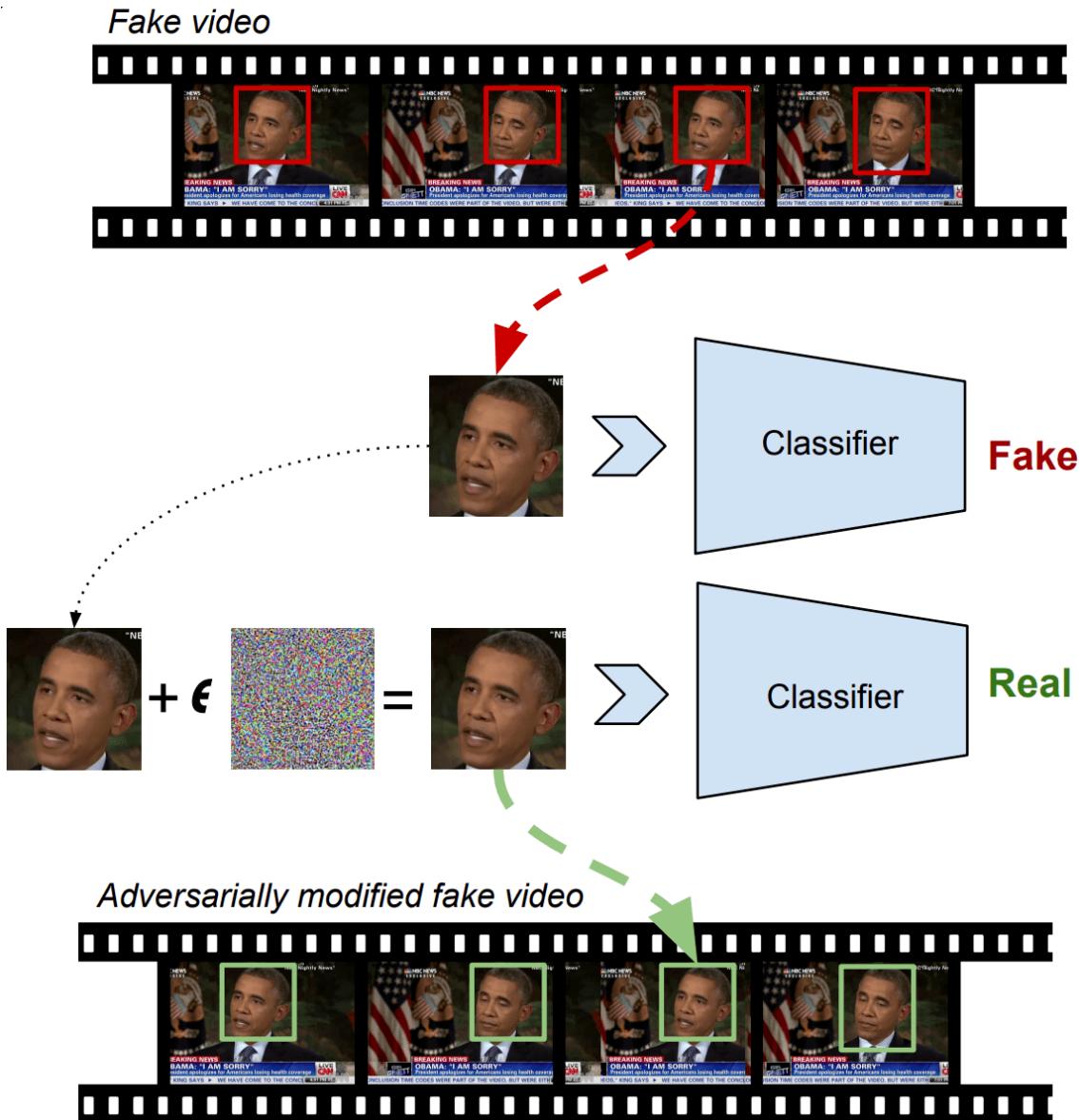


Figura 5.5: Evasión de defensas de *deepfakes* mediante Adversarial DeepFakes. Fuente: [Paarth Neekhara et al.](#)

adversarios (en imágenes) son más sensibles a cierto tipos de transformaciones como rotación y desplazamiento que las imágenes normales.

- **Enmascarado/regularización del gradiente:** este método esconde los gradientes o reduce la sensibilidad de los modelos. En este sentido se han propuesto defensas como [Madry et al.](#), [Song et al.](#) o [Ma et al.](#), entre otras. Este tipo de defensas **no parecen ser muy efectivas**. De hecho, [Athalye et al.](#) son capaces de saltarse los mecanismos de defensa de todas estas defensas de forma muy eficaz (hasta la fecha de publicación del paper). La tendencia actual es que cada vez que se propone una defensa de este tipo, pasa poco tiempo hasta que se consigue romper. De hecho, se produce un

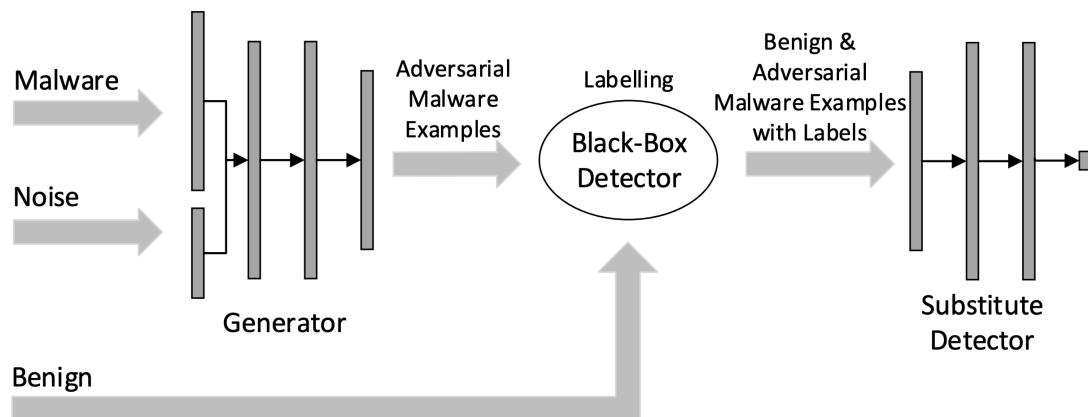


Figura 5.6: Arquitectura de MalGAN. Fuente: [Weiwei Hu, Ying Tan](#).

tira y afloja entre atacantes y defensores, y los atacantes parecen ir por delante por el momento, aunque se siguen proponiendo defensas cada vez más difíciles de evadir.

- **Distillation:** Papernot et al. propusieron esta defensa para evitar los ataques de los métodos FGSM y JSMA.
- **Redes de defensa:** introducir otro tipo de redes neuronales para defenderte de este tipo de ataques. Por ejemplo, Gu et al. usa redes profundas contractivas con autoencoders contractivos y *denoising autoencoders*, que puede eliminar ruido de los ejemplos adversarios.
- **Defensas débiles:** la defensa se basa en el hecho de que un *ensemble* de muchas **defensas débiles**, que no son suficientes por sí mismas. Un ejemplo de ello es *Athena*, framework para defender sistemas de *deep learning* frente a ejemplos adversarios en modelos basados en imágenes. Se basa en que la transformación (rotación, desplazamiento, etc.) de los datos de entrada entrenadas en conjuntos de datos disjuntos, previamente transformados supone una defensa frente a ataques adversarios.

Athena emplea cinco estrategias de ensamblado de las defensas débiles:

- **Defensa aleatoria:** selecciona una defensa débil aleatoriamente para que ella sea la que devuelva la predicción.
- **Mayoría de voto** (MV): recolecta todas las clases de las etiquetas y devuelve la que con mayor frecuencia se ha devuelto por las defensas débiles.
- **Mayoría de votos del top 2:** similar a MV, pero recolecta las etiquetas asociadas al *top 2* con mayor probabilidad, y se realiza la mayoría del voto entre ellas.

- **Probabilidad media:** produce la respuesta final como la media de las probabilidades de las defensas débiles.
- Logits promedio: utiliza la información almacenada en los *logits* para devolver la etiqueta con la media de *logits* más altos.

5.4 Herramientas

Numerosas herramientas *opensource* se pueden encontrar que implementen ataques y defensas de evasión y permitan robustecer los modelos de *machine learning*.

5.4.1 Cleverhans

[Cleverhans](#) es una librería para construir ataques adversarios y medir cómo de robusto es un modelo de *deep learning* en modelos de imágenes. Está desarrollado en Python y se integra con los frameworks Tensorflow, Torch y JAX. Implementa numerosos ataques como L-BFGS, FGSM, JSMA, C&W, entre otros. Tiene pensado añadir en el futuro defensas, aunque a día de hoy no están implementadas en el proyecto.

Esta librería dispone de amplia documentación, tutoriales y ejemplos fáciles de entender. El proyecto se encuentra dentro de la organización de [Tensorflow](#) en GitHub, por lo que es esperable que se mantenga el soporte y se añadan nuevas funcionalidades, según vaya avanzando el estado del arte.

5.4.2 Advertorch

[Advertorch](#) es una librería desarrollada en Python para comprobar la robustez de modelos de *deep learning* basados en imágenes frente a ataques con ejemplos adversarios. Esta librería solamente soporta PyTorch. Implementa numerosos ataques vistos anteriormente. Asimismo, implementa defensas basadas en la transformación de imágenes como compresión en JPEG (elimina ruido adversario en las imágenes) o distintos filtros que aplicar a las imágenes como [filtros gaussianos](#) o de suavizado.

Esta librería actualmente se encuentra en la versión 0.2 y parece muy prometedora. Es esperable que, si sigue con su desarrollo, se convierta en una de las librerías de referencia en este ámbito, aunque estará lastrada a sólo poder emplearse con modelos de PyTorch.

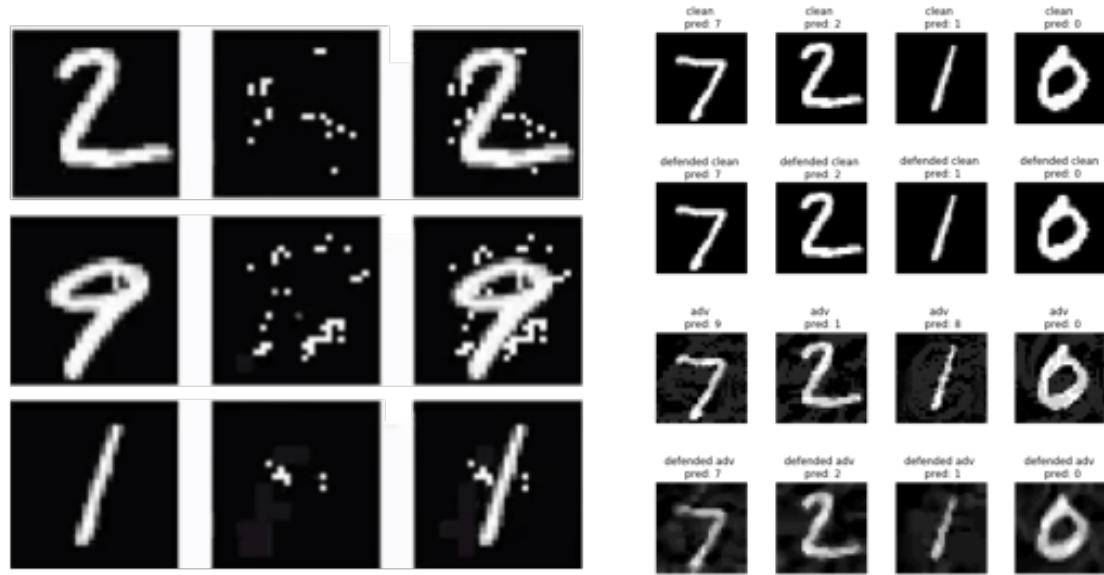


Figura 5.7: Ejemplos adversarios generados con Cleverhans y Advertorch, respectivamente sobre el conjunto de datos del [MNIST](#).

5.4.3 Foolbox

[Foolbox](#) es un conjunto de herramientas para engañar redes neuronales usando ejemplos adversarios. Está implementada en Python y requiere Numpy y Scipy para funcionar. Soporta muchos frameworks de *deep learning* como Tensorflow, Keras, PyTorch, JAX, Theano, MXNet y [Lasagne](#). Implementa muchos de los ataques descritos en la sección de ataques y dispone de una gran documentación con numerosos ejemplos.

5.4.4 IBM Adversarial Robustness Toolbox (ART)

[ART](#) [10] es una librería desarrollada en Python para la defensa de algoritmos de *machine learning*. Permite la defensa de redes neuronales, SVM, árboles de decisión, regresión logística, etc. Además, es de propósito general, es decir, no se centra en ataques adversarios solamente sino también en el resto de ataques que pueden sufrir este tipo de algoritmos: evasión, envenenamiento, extracción e inversión. Implementa un gran número de ataques así como muchas defensas. Soporta muchos frameworks como Tensorflow, Keras, PyTorch, Scikit learn, etc.

ART es un proyecto muy activo y con un continuo desarrollo. Esto hace que sea de las mejores herramientas actuales para comprobar la seguridad de modelos de *machine learning*. Ver Sección 6 para ver más detalles y un tutorial de esta herramienta.

5.4.5 Artificial Adversary

[Artificial Adversary](#) es una herramienta escrita en Python que genera ejemplos adversarios en texto y comprueba la robustez de los mismos. Desarrollado por Airbnb. Proporciona ataques en texto y algún ejemplo, aunque insuficiente documentación. El proyecto no parece estar activo ya que su último *commit* es de agosto de 2018.

5.4.6 AdvBox

[AdvBox](#) es un conjunto de herramientas escritas en Python para generar ejemplos adversarios para engañar redes neuronales desarrollada por [Baidu](#). Soporta los frameworks como [PaddlePaddle](#), PyTorch, Caffe2, MxNet, Keras y Tensorflow. AdvBox provee una CLI para generar ejemplos adversarios sin necesidad de escribir una sola línea de código. Implementa numerosos ataques y defensas. No cuenta con mucha documentación, pero parece suficiente para entender el funcionamiento de este proyecto.

5.4.7 DeepRobust

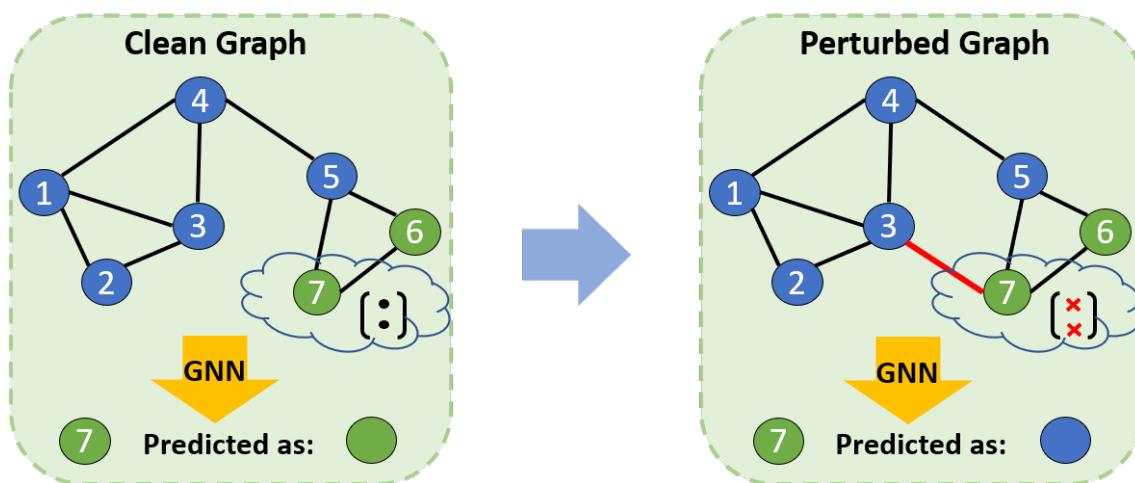


Figura 5.8: Ejemplo adversario sobre un grafo. Fuente: DeepRobust.

[DeepRobust](#) es una librería para Python de ataque y defensa de adversarios para imágenes y grafos para PyTorch. En cuanto a imágenes, implementa los ataques típicos como FGSM, L-BFGS y C&W, entre otros, y como defensa implementa diferentes técnicas de entrenamiento adversario. Lo más relevante de esta herramienta es que implementa ataques y defensas sobre [grafos](#) (Figura 5.8).

Los ejemplos adversarios sobre grafos tratan de modificar un grafo de entrada cambiando unas pocas aristas y características haciendo que se clasifique de

forma incorrecta.

Los ataques y defensas que implementa DeepRobust se pueden encontrar descritos en los artículos de [Jin et al.](#) y [Xu et al.](#)

DeepRobust es una librería con un desarrollo activo y puede convertirse en una de las herramientas de referencia en la implementación de ataques y defensas sobre [redes neuronales de grafos](#).

6 Tutorial: Adversarial Robustness Tool (ART)

Adversarial Robustness Tool [10], abreviado como ART, es una librería opensource de Adversarial Machine Learning que permite comprobar la robustez de los modelos de *machine learning*. Está desarrollada en Python e **implementa ataques y defensas de extracción, inversión, envenenamiento y evasión**. ART soporta los frameworks más populares: Tensorflow, Keras, PyTorch, MxNet, Scikit-Learn, entre muchos otros). Además, **no está limitada al uso de modelos que emplean imágenes como entrada**, sino que soporta otros tipos de datos como audio, vídeo, datos tabulares, etc.



Figura 6.1: Logo de Adversarial Robustness Tool.

ART es una librería desarrollada por IBM, y que recientemente ha pasado a [formar parte de las herramientas de Linux Foundation AI](#). ART es un proyecto muy activo que cuenta con amplia [documentación](#) y [ejemplos](#) para probar de forma sencilla y rápida distintos algoritmos estado del arte en ataque y defensa. En el momento de escribir estas líneas, la última versión estable es la **1.4.1**, que emplearemos a lo largo del tutorial.

La librería **se organiza en distintos paquetes Python** que ordenan los distintos algoritmos implementados. Entre los módulos más empleados se encuentran **art.attacks** y **art.defences**, que incluyen la mayoría de ataques y defensas, respectivamente. Estos módulos, a su vez, se subdividen en otros, que ordenan los ataques y defensas por tipo. En el caso **art.attacks** tiene como submódulos **art.attacks.{extraction, model_inversion, poisoning, evasion}**. De forma similar, se hace con el módulo de defensas. Esto permite localizar fácilmente los ataques y defensas disponibles para cada tipo específico.

6.1 Prerrequisitos e instalación

Para instalar ART sólo se requiere tener instalada la versión 3 de **Python** y **pip**.

La instalación se realiza a través de **pip** del siguiente modo.

```
pip install adversarial-robustness-toolbox==1.4.1
```

Se recomienda el uso de GPU para acelerar los cálculos, aunque se puede realizar perfectamente con sólo CPU. En caso de no disponer de GPU propia, se puede utilizar el servicio [Colab](#) de Google, que proporciona acceso a GPUs de forma gratuita.

6.2 ¡Hola, ART!

A lo largo de todo el tutorial, nos centraremos en ataques de caja blanca que nos permitan robustecer nuestros modelos aplicando defensas, pero ART permite realizar ataques de caja negra sin ningún tipo de limitación.

Para comprobar que la instalación se ha realizado correctamente, vamos a probar un ataque de evasión. En este ejemplo, necesitaremos instalar las dependencias de **Keras** y **Tensorflow**, ya que emplearemos **Keras** para entrenar los modelos y **Tensorflow** será el *backend*. Todo el código se encuentra disponible en <https://github.com/bbvanexttechnologies/art-tutorial>.

```
pip install keras tensorflow-gpu # GPU  
pip install keras tensorflow      # CPU
```

El ataque que emplearemos será FGSM, un ataque de evasión de caja blanca sobre el conjunto de datos del MNIST. Este conjunto datos permite clasificar dígitos manuscritos del 0 al 9 (Figura 6.2). Se compone de 60 000 imágenes de 28x28 píxeles que representan dígitos manuscritos, de los que 50 000 se emplean como datos de entrenamiento y los 10 000 como datos de *testing*.

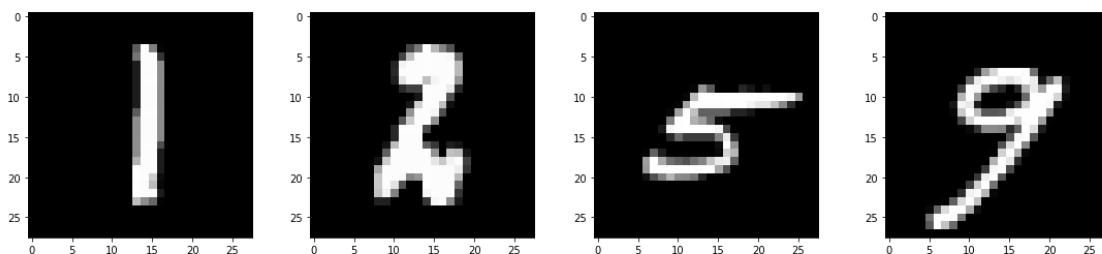


Figura 6.2: Algunas muestras del conjunto de datos del MNIST.

Comenzamos importando las dependencias necesarias en un fichero. Las primeras corresponden a **Keras**, que permiten crear el modelo al que atacar y las últimas corresponden al módulo de ART que nos permitirán generar ejemplos adversario para el modelo.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
import numpy as np
import matplotlib.pyplot as plt

from art.attacks.evasion import FastGradientMethod
from art.estimators.classification import KerasClassifier
from art.utils import load_mnist
```

El siguiente paso es cargar el conjunto de datos y entrenar la arquitectura del modelo. En este ejemplo se usan 3 épocas de entrenamiento que son suficientes para obtener una alta precisión.

```
(x_train, y_train), (x_test, y_test), min_pixel_value,
↪ max_pixel_value = load_mnist()

model = Sequential()
model.add(Conv2D(filters=4, kernel_size=(5, 5), strides=1,
↪ activation="relu", input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(filters=10, kernel_size=(5, 5), strides=1,
↪ activation="relu", input_shape=(23, 23, 4)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(100, activation="relu"))
model.add(Dense(10, activation="softmax"))

model.compile(
loss=keras.losses.categorical_crossentropy,
↪ optimizer=keras.optimizers.Adam(lr=0.01),
↪ metrics=[ "accuracy" ]
)
classifier = KerasClassifier(model=model,
↪ clip_values=(min_pixel_value, max_pixel_value),
↪ use_logits=False)

classifier.fit(x_train, y_train, batch_size=64, nb_epochs=3)
```

Calculamos la precisión del modelo para ver cómo de bueno fue el entrenamiento. En nuestro caso es del **97.86%**.

```

predictions_test = classifier.predict(x_test)
accuracy = np.sum(np.argmax(predictions_test, axis=1) ==
                  np.argmax(y_test, axis=1)) / len(y_test)
print("Accuracy on test examples: {}%".format(accuracy * 100))

```

Por último, realizamos el ataque FGSM al modelo que acabamos de entrenar. Fijamos el parámetro **eps** como 0.2 y generamos ejemplos adversarios para el conjunto de *test* para comprobar cómo varía la precisión con respecto al modelo original, que resulta ser del 39.93%, una bajada bastante significativa.

```

attack = FastGradientMethod(estimator=classifier, eps=0.2)
x_test_adv = attack.generate(x=x_test)

predictions = model.predict(x_test_adv)
accuracy = np.sum(np.argmax(predictions, axis=1) ==
                  np.argmax(y_test, axis=1)) / len(y_test)
print("Accuracy on adversarial test examples:
      {}%".format(accuracy * 100))

```

Veamos algún ejemplo adversario que se genera con este método. En este caso, sólo mostramos la muestra 1234 del conjunto de *test*, que el modelo original lo clasifica de forma correcta como 8, sin embargo, el ejemplo adversario generado es clasificado como 3 (Figura 6.3).

```

sample = 1234

fig = plt.figure()
ax = fig.add_subplot(1, 2, 1)
ax.imshow(x_test[sample], cmap='gray', interpolation='none')
ax.set_title("Original: {}".format(np.argmax(y_test[sample])))
ax.axis('off')
ax = fig.add_subplot(1, 2, 2)
ax.imshow(x_test_adv[sample], cmap='gray', interpolation='none')
ax.set_title("Adversarial:
      {}".format(np.argmax(predictions[sample])))
ax.axis('off')

```

6.3 Ataques de evasión

Continuamos con los ataques de evasión viendo algún método más de caja blanca. Aquí, generaremos ejemplos adversarios con los métodos JSMA y C&W L_2 (ver detalles en la Sección 5.2).

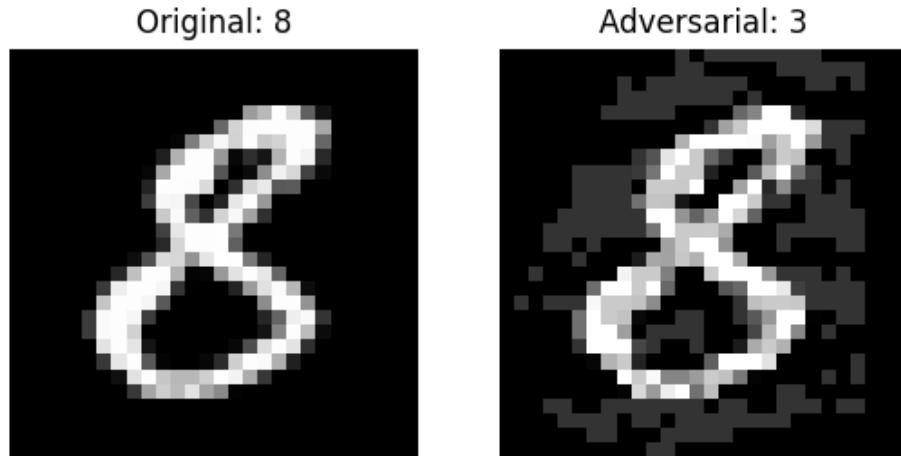


Figura 6.3: Ejemplo adversario generado con el método FGSM.

```
from art.attacks.evasion import SaliencyMapMethod, CarliniL2Method
```

Y de igual forma, que en ejemplo previo, generamos los ejemplos adversarios. En este caso fijamos el parámetro `theta=0.1` para JSMA y dejamos los parámetros por defecto en C&W L₂. Este proceso tarda bastante tiempo, aún usando GPUs. En este caso, se obtiene una precisión de 0.42% para JSMA y 84.15% para C&W L₂. En nuestro caso, JSMA es un ataque muy eficiente, sin embargo C&W L₂ no afecta apenas nada a este modelo. Cambiando con los parámetros de los distintos métodos se pueden obtener resultados distintos.

```
attack_jsma = SaliencyMapMethod(classifier = classifier, theta =
    ↵ 0.1)
x_test_jsma = attack_jsma.generate(x=x_test)

attack_cw2 = CarliniL2Method(classifier = classifier)
x_test_cw2 = attack_cw2.generate(x=x_test)

attacks = ["fgm", "jsma", "c&w l2"]
x_tests_attacks = [x_test_fgm, x_test_jsma, x_test_cw2]
predictions = []

for i, x in enumerate(x_tests_attacks):
    preds = classifier.predict(x)
    predictions.append(preds)
    accuracy = np.sum(np.argmax(preds, axis=1) == np.argmax(y_test,
        ↵ axis=1)) / len(y_test)
    print("Accuracy on adversarial test examples for {} attack:
        ↵ {}%".format(attacks[i], accuracy * 100))
```

Veamos algunos de los ejemplos generados por cada uno de los métodos, incluido. En el caso de la muestra 1234 (Figura 6.4), el modelo clasifica la entrada del

método FGSM como un 3, con JSMA un 5 y con C&W L₂ clasifica de forma correcta. Este resultado C&W L₂ viene a confirmar los malos resultados sobre el conjunto de *test*.

```
sample = 1234
fig = plt.figure(figsize=(20,10))
ax = fig.add_subplot(1, 4, 1)
ax.imshow(x_test[sample].reshape((28, 28)), cmap='gray',
          interpolation='none')
ax.set_title("Original: {}".format(np.argmax(y_test[sample])))
ax.axis('off')
ax = fig.add_subplot(1, 4, 2)
ax.imshow(x_test_fgm[sample].reshape((28, 28)), cmap='gray',
          interpolation='none')
ax.set_title("Adversarial (FGSM):"
             " {}".format(np.argmax(predictions[0][sample])))
ax.axis('off')
ax = fig.add_subplot(1, 4, 3)
ax.imshow(x_test_jsma[sample].reshape((28, 28)), cmap='gray',
          interpolation='none')
ax.set_title("Adversarial (JSMA):"
             " {}".format(np.argmax(predictions[1][sample])))
ax.axis('off')
ax = fig.add_subplot(1, 4, 4)
ax.imshow(x_test_cw2[sample].reshape((28, 28)), cmap='gray',
          interpolation='none')
ax.set_title("Adversarial (C&W L2):"
             " {}".format(np.argmax(predictions[2][sample])))
ax.axis('off')
fig.show()
```

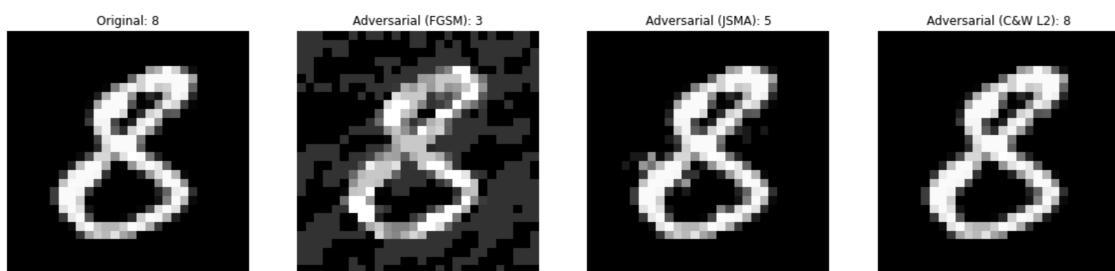


Figura 6.4: Ejemplo de adversarios generados con los métodos FGSM, JSMA y C&W L₂.

Más ataques se pueden encontrar en el módulo [art.attacks.evasion](#).

6.3.1 Defensa: entrenamiento adversario

Una defensa sencilla de implementar es el entrenamiento adversario (*adversarial training* en inglés), que consiste en entrenar el modelo empleando ejemplos

adversarios generados previamente con alguno de los métodos.

En este caso, generamos ejemplos adversarios sobre los datos de entrenamiento con el método FGSM, que es más rápido, pero se puede usar cualquier otro o incluso una combinación de varios métodos para hacer un modelo más robusto.

```
x_train_fgm = attack_fgm.generate(x=x_train)
```

Estos nuevos datos se incorporan al conjunto de entrenamiento junto con sus respectivas salidas.

```
x_train = np.append(x_train, x_train_fgm, axis=0)
y_train = np.append(y_train, y_train, axis=0)
```

Sólo queda entrenar el modelo desde cero y comprobar cuál es su precisión y compararla con el entrenamiento sin emplear entrenamiento adversario.

```
predictions_test = classifier.predict(x_test)
accuracy = np.sum(np.argmax(predictions_test, axis=1) ==
                  np.argmax(y_test, axis=1)) / len(y_test)
print("Accuracy on test examples: {}%".format(accuracy * 100))

predictions_fgm = classifier.predict(x_test_fgm)
accuracy = np.sum(np.argmax(predictions_fgm, axis=1) ==
                  np.argmax(y_test, axis=1)) / len(y_test)
print("Accuracy on adversarial test examples for FGSM attack:
      {}%".format(accuracy * 100))
```

En nuestro caso, se obtiene una precisión del 96.14% utilizando entrenamiento adversario, que no está muy lejos del entrenamiento realizado sin ejemplos adversarios, que era de un 97.86% y por muy encima de emplear ejemplos adversarios durante la evaluación (39.93%). Esto demuestra que una defensa muy básica como esta puede resultar muy útil en caso de un ataque de evasión.

Otra forma de realizar entrenamiento adversario es hacer uso de la clase nativa de ART llamada **AdversarialTrainer** del módulo **art.defences.trainer**. Esta defensa no es exactamente igual al anterior ya que emplea un *ensemble* de modelos.

```

from art.defences.trainer import AdversarialTrainer

model.compile(loss=keras.losses.categorical_crossentropy,
    ↪ optimizer=keras.optimizers.Adam(lr=0.01),
    ↪ metrics=[ "accuracy" ])

defence = AdversarialTrainer(classifier=classifier,
    ↪ attacks=attack_fgm, ratio=0.5)

# Recuperamos los datos originales
(x_train, y_train), (x_test, y_test), min_pixel_value,
    ↪ max_pixel_value = load_mnist()

defence.fit(x=x_train, y=y_train, nb_epochs=3)

predictions_test = classifier.predict(x_test)
accuracy = np.sum(np.argmax(predictions_test, axis=1) ==
    ↪ np.argmax(y_test, axis=1)) / len(y_test)
print("Accuracy on test examples: {}%".format(accuracy * 100))

predictions_fsm = defence.predict(x_test_fgm)
accuracy = np.sum(np.argmax(predictions_fsm, axis=1) ==
    ↪ np.argmax(y_test, axis=1)) / len(y_test)
print("Accuracy on adversarial test examples for JSM attack:
    ↪ {}%".format(accuracy * 100))

```

En este caso, se obtiene una precisión del 94.39% empleando esta defensa, que no es muy diferente a la obtenida anteriormente con el método anterior.

Los módulos `art.defences.detector.evasion` y `art.defences.trainer` contienen muchas más defensas.

6.4 Ataques de inversión

ART implementa ataques de inversión en el módulo `art.attacks.inference`. A su vez, se subdivide en ataques Membership Inference Attacks (MIA) que se encuentran en el módulo `art.attacks.inference.membership_inference`, Property Inference Attacks (PIA) en el módulo `art.attacks.inference.attribute_inference` y los ataques de reconstrucción en el módulo `art.attacks.inference.model_inversion`.

Por brevedad, sólo nos centraremos en el ataque de reconstrucción de Fredrikson et al. [3]. ART implementa este ataque a través de la clase `MIFace`, y aunque el ataque se propuso para explotar sistemas de reconocimiento facial, se ha demostrado efectivo como se comenta en la [documentación de ART](#) y como veremos a continuación. La mayor parte del código ha sido tomado del [ejemplo](#) que proporciona ART en su repositorio de GitHub.

El primer paso, de nuevo, consiste en entrenar un modelo similar a los anteriores empleando los datos del MNIST. Este modelo difiere de los anteriores en los filtros de las capas convolucionales, la capa de Dropout y el número de épocas (de 3 a 10) del entrenamiento.

```
# [imports omitidos por brevedad]
from keras.layers import Dropout

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
    ↴ input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
    ↴ metrics=['accuracy'])

classifier = KerasClassifier(model=model,
    ↴ clip_values=(min_pixel_value, max_pixel_value),
    ↴ use_logits=False)

classifier.fit(x_train, y_train, batch_size=128, nb_epochs=10)
```

El clasificador logra una precisión del **98.49%** sobre el conjunto de *test*. El ataque se realiza empleando la clase **MIFace**, empleando como clasificador el modelo que acabamos de entrenar.

```
from art.attacks.inference.model_inversion import MIFace

y = np.arange(10)
attack = MIFace(classifier, max_iter=10000, threshold=0.99)
```

Para favorecer a converger al ataque se puede emplear, de forma opcional, un vector de inicialización. En este caso, empleamos los descritos en el [ejemplo de ART](#). Estos son inicializar la imagen con los píxeles en blanco, en gris, en negro, de forma aleatoria y con la media de los valores del conjunto de *test*.

```

x_init_white = np.zeros((10, 28, 28, 1))
x_init_grey = np.zeros((10, 28, 28, 1)) + 0.5
x_init_black = np.ones((10, 28, 28, 1))
x_init_random = np.random.uniform(0, 1, (10, 28, 28, 1))
x_init_average = np.zeros((10, 28, 28, 1)) + np.mean(x_test,
    axis=0)

x_inits = [x_init_white, x_init_grey, x_init_black,
    x_init_random, x_init_average]
x_inits_names = ["white", "grey", "black", "random", "average"]

x_infers = []
for x in x_inits:
    x_infers.append(attack.infer(x, y))

```

Sólo queda ver qué ha sido capaz de inferir este ataque para cada vector de inicialización.

```

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(7, 15))
outer = gridspec.GridSpec(5, 1, hspace=0.5)

for i, x_infer in enumerate(x_infers):
    inner = gridspec.GridSpecFromSubplotSpec(2, 5,
        subplot_spec=outer[i], wspace=0.1, hspace=0.1)

    for j in range(10):
        ax = plt.Subplot(fig, inner[j])
        ax.imshow(x_infers[i][j].reshape((28, 28)), cmap='gray_r')

        if j == 2:
            ax.set_title(x_inits_names[i])
            ax.set_xticks([])
            ax.set_yticks([])
            fig.add_subplot(ax)

    fig.show()

```

La Figura 6.5 muestra que los mejores vectores de inicialización son el gris y la media. De hecho, en estos últimos, se puede apreciar que las muestras inferidas por el ataque son muy similares a las que proporciona el MNIST.

A fecha de escribir de escribir este tutorial no existen defensas implementadas en ataques de inversión, pero es esperable que en el futuro no muy lejano se implementen en ART.

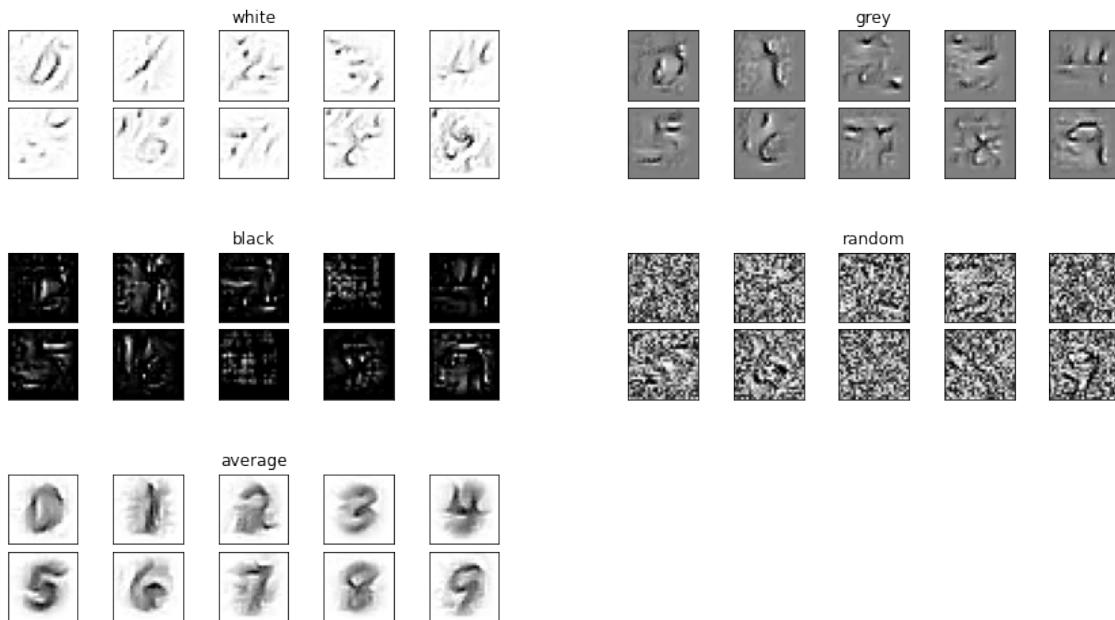


Figura 6.5: Filtrado de información del conjunto de entrenamiento con distintos vectores de inicialización.

6.5 Ataques de envenenamiento

El módulo `art.attacks.poisoning` de ART implementa algunos ataques de envenenamiento, que permite introducir una puerta trasera en un modelo, construyendo una BadNet [4] y por último, cómo es posible defenderse frente a este tipo de ataques.

En este caso, necesitamos instalar versiones específicas de **Tensorflow** y **Keras** para que funcione correctamente los ejemplos.

```
pip install keras==2.2.4 tensorflow-gpu==1.9.0 # GPU
pip install keras==2.2.4 tensorflow==1.9.0      # CPU
```

Con todo lo anterior instalado, podemos comenzar a crear una puerta trasera sobre el conjunto de datos del MNIST. De nuevo, nos basamos en el [ejemplo de envenenamiento](#) que se encuentra en el repositorio de ART en GitHub. Crearemos una puerta trasera, que al detectar un patrón de píxeles en la esquina superior derecha, clasifique cada dígito como el dígito siguiente, es decir, el 0 como 1, el 1 como 2, hasta el 9 como 0.

Lo primero de todo es cargar el conjunto de datos del MNIST. En esta ocasión, cargamos los datos en crudo empleando el parámetro `raw = True`.

```
(x_raw, y_raw), (x_raw_test, y_raw_test), min_, max_ =
    load_mnist(raw=True)
```

De todos los datos, seleccionamos 7500 de forma aleatoria, en los que se producirá el envenenamiento. Un adversario intentaría envenenar la menor cantidad de datos posibles para evitar ser detectado.

```
n_train = np.shape(x_raw)[0]
num_selection = 7500
random_selection_indices = np.random.choice(n_train,
    num_selection)
x_raw = x_raw[random_selection_indices]
y_raw = y_raw[random_selection_indices]
```

Definimos la función que se encargará de introducir el patrón de píxeles en la esquina inferior derecha de la imagen, que se encuentra implementada ya en ART. Este patrón se puede ver en la Figura 6.6.

```
from art.attacks.poisoning.perturbations import add_pattern_bd

max_val = np.max(x_raw)
def poison_func(x):
    return add_pattern_bd(x, pixel_value=max_val)
```



Figura 6.6: Patrón de píxeles introducida en distintas imágenes.

La función más importante es la que envenena el conjunto de datos seleccionado, introduciendo la puerta trasera llamada **poison_dataset**. En ella, se emplea la clase **PoisoningAttackBackdoor**, que implementa el ataque de BadNets [4].

```

from art.attacks.poisoning import PoisoningAttackBackdoor
from art.utils import preprocess

def poison_dataset(x_clean, y_clean, percent_poison, poison_func):
    x_poison = np.copy(x_clean)
    y_poison = np.copy(y_clean)
    is_poison = np.zeros(np.shape(y_poison))

    sources = np.arange(10) # 0, 1, 2, 3, ...
    targets = (np.arange(10) + 1) % 10 # 1, 2, 3, 4, ...
    for i, (src, tgt) in enumerate(zip(sources, targets)):
        n_points_in_tgt = np.size(np.where(y_clean == tgt))
        num_poison = round((percent_poison * n_points_in_tgt) / (1
            - percent_poison))
        src_imgs = x_clean[y_clean == src]

        n_points_in_src = np.shape(src_imgs)[0]
        indices_to_be_poisoned = np.random.choice(n_points_in_src,
            num_poison)

        imgs_to_be_poisoned =
            np.copy(src_imgs[indices_to_be_poisoned])
        backdoor_attack = PoisoningAttackBackdoor(poison_func)
        imgs_to_be_poisoned, poison_labels =
            backdoor_attack.poison(imgs_to_be_poisoned,
            y=np.ones(num_poison) * tgt)
        x_poison = np.append(x_poison, imgs_to_be_poisoned, axis=0)
        y_poison = np.append(y_poison, poison_labels, axis=0)
        is_poison = np.append(is_poison, np.ones(num_poison))

    is_poison = is_poison != 0
    return is_poison, x_poison, y_poison

percent_poison = .33
(is_poison_train, x_poisoned_raw, y_poisoned_raw) =
    poison_dataset(x_raw, y_raw, percent_poison, poison_func)
x_train, y_train = preprocess(x_poisoned_raw, y_poisoned_raw)
x_train = np.expand_dims(x_train, axis=3)

(is_poison_test, x_poisoned_raw_test, y_poisoned_raw_test) =
    poison_dataset(x_raw_test, y_raw_test, percent_poison,
    poison_func)
x_test, y_test = preprocess(x_poisoned_raw_test,
    y_poisoned_raw_test)
x_test = np.expand_dims(x_test, axis=3)

n_train = np.shape(y_train)[0]
shuffled_indices = np.arange(n_train)
np.random.shuffle(shuffled_indices)
x_train = x_train[shuffled_indices]
y_train = y_train[shuffled_indices]

```

Se aplica la función anterior sobre los conjuntos de entrenamiento y *test* para

evaluar (reintroduciendo los nuevos datos envenenados de forma aleatoria). Ya sólo queda entrenar el modelo y evaluar cómo se comporta el modelo.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
    ↵ input_shape=x_train.shape[1:]))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
    ↵ metrics=['accuracy'])

classifier = KerasClassifier(model=model, clip_values=(min_,
    ↵ max_))
classifier.fit(x_train, y_train, nb_epochs=3, batch_size=128)
```

```
clean_x_test = x_test[is_poison_test == 0]
clean_y_test = y_test[is_poison_test == 0]

clean_preds = np.argmax(classifier.predict(clean_x_test), axis=1)
clean_correct = np.sum(clean_preds == np.argmax(clean_y_test,
    ↵ axis=1))
clean_total = clean_y_test.shape[0]

clean_acc = clean_correct / clean_total
print("Clean test set accuracy: {:.2f}%".format(clean_acc * 100))

poison_x_test = x_test[is_poison_test]
poison_y_test = y_test[is_poison_test]

poison_preds = np.argmax(classifier.predict(poison_x_test),
    ↵ axis=1)
poison_correct = np.sum(poison_preds == np.argmax(poison_y_test,
    ↵ axis=1))
poison_total = poison_y_test.shape[0]

poison_acc = poison_correct / poison_total
print("Effectiveness of poison: {:.2f}%".format(poison_acc * 100))
```

El modelo entrenado tiene una precisión sobre el conjunto de *test* sin envenenar del 96.75% y la puerta trasera funciona en el 95.01% de los casos, por lo que se ha conseguido introducir la BadNet de forma satisfactoria.

Representamos cómo se comporta la puerta trasera sobre una imagen legítima y sobre una imagen con el disparador de la puerta trasera (el patrón de píxeles

en la esquina inferior derecha). En la Figura 6.7 se puede ver una instancia que es clasificada por el modelo (izquierda), pero que al introducir una muestra que contiene el disparador se clasifica como en número siguiente del que debería ser, esto es, 2 en este caso (derecha).

```
c = 2 # class to display
i = 0 # image of the class to display

c_idx = np.where(np.argmax(clean_y_test,1) == c)[0][i]
c_idx_p = np.where(np.argmax(poison_y_test,1) == c)[0][i]

fig = plt.figure(figsize=(20,10))
ax = fig.add_subplot(1, 2, 1)
ax.imshow(clean_x_test[c_idx].reshape((28, 28)), cmap="gray")
ax.axis('off')
ax.set_title("Prediction: {}".format(clean_preds[c_idx]))
ax = fig.add_subplot(1, 2, 2)
ax.imshow(poison_x_test[c_idx_p].reshape((28, 28)), cmap="gray")
ax.axis('off')
ax.set_title("Prediction: {}".format(poison_preds[c_idx_p]))
plt.show()
```

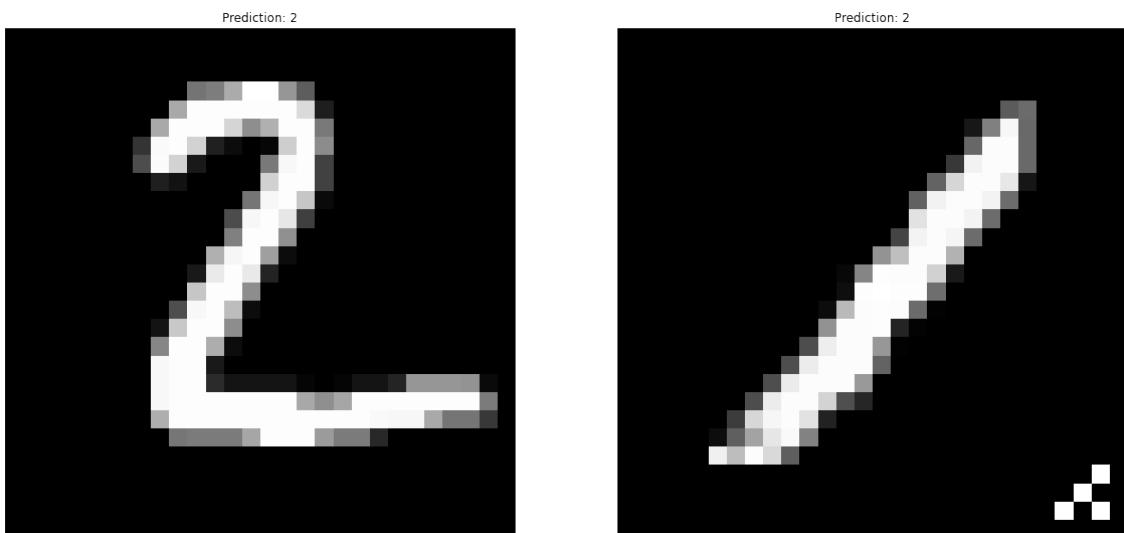


Figura 6.7: Ejemplo del correcto funcionamiento de la puerta trasera. A la izquierda se muestra una muestra que es clasificada correctamente por el modelo y, a la derecha, una muestra que se clasifica de acuerdo a la puerta trasera resultado de envenenar el conjunto de datos.

6.5.1 Defensa: Neural Cleanse

Neural Cleanse es una defensa que permite identificar puertas traseras, reconstruir disparadores e incluye distintas mitigaciones. ART implementa esta defensa en la clase **NeuralCleanse** del módulo `art.defences.transformer.poisoning`.

A fecha de escribir este tutorial, Neural Cleanse sólo soporta la versión **2.2.4** de Keras, por lo que esto explica por qué había que instalar versiones específicas de Keras y Tensorflow.

Lo primero es crear una instancia de esta clase para crear la defensa y pasar el clasificador que queremos defender.

```
from art.defences.transformer.poisoning import NeuralCleanse

cleanse = NeuralCleanse(classifier)
defence_cleanse = cleanse(classifier, steps=10, learning_rate=0.1)
```

Con Neural Cleanse, se puede detectar la puerta trasera para cada una de las clases, aunque sólo la aplicaremos en la clase 1, por brevedad.

```
pattern, mask = defence_cleanse.generate_backdoor(x_test, y_test,
    ↵ np.array([0, 1, 0, 0, 0, 0, 0, 0, 0, 0]))
plt.imshow(np.squeeze(mask * pattern))
plt.axis("off")
```

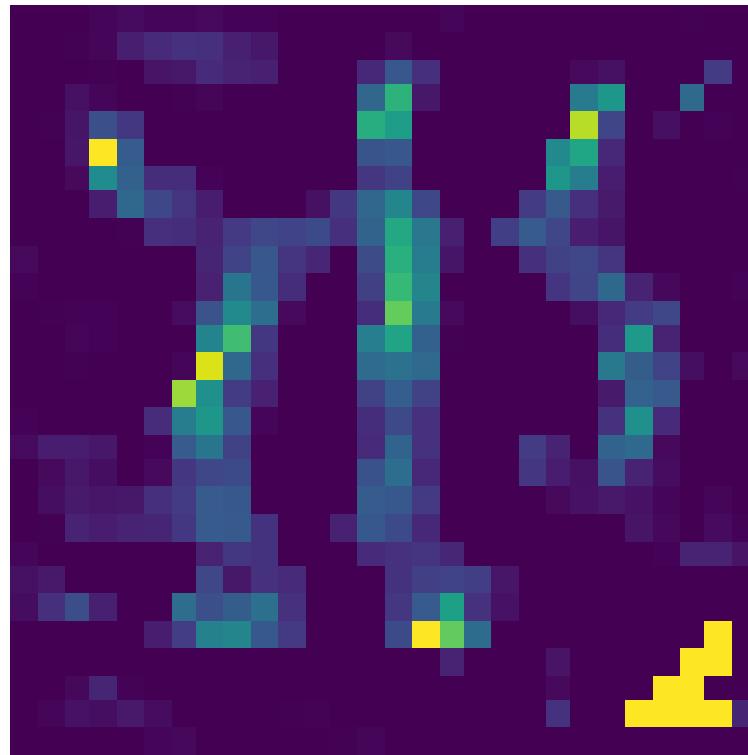


Figura 6.8: Detección del patrón de la puerta trasera para la clase 1 con Neural Cleanse.

En la Figura 6.8 se puede ver cómo este método detecta la puerta trasera en la

esquina inferior derecha. Habiendo visto que se detecta la puerta trasera se pueden aplicar mitigaciones como filtrado, desaprendizaje y poda.

- Filtrado (*filtering* en inglés) que detecta y bloquea instancias consideradas amenazas durante la fase de inferencia.
- Desaprendizaje (*unlearning* en inglés), que consiste en reentrenar las puertas traseras con las etiquetas legítimas.
- Poda (*pruning* en inglés), que consiste en eliminar las neuronas asociadas a la puerta trasera.

```
defence_cleanse.mitigate(clean_x_test, clean_y_test,  
    ↵ mitigation_types=[ "filtering" ])  
  
defence_cleanse.mitigate(clean_x_test, clean_y_test,  
    ↵ mitigation_types=[ "unlearning" ])  
  
defence_cleanse.mitigate(clean_x_test, clean_y_test,  
    ↵ mitigation_types=[ "pruning" ])
```

En nuestro caso, se obtiene que el filtrado consigue filtrar un **40.89%** de las muestras maliciosas, el desaprendizaje rebaja hasta el **30.03%** la efectividad de envenenamiento, mientras que la poda hace que la efectividad sea del **0.00%**, pero hace que la precisión del modelo sea del **46.30%**.

Las mitigaciones también se pueden aplicar de forma conjunta.

```
defence_cleanse.mitigate(clean_x_test, clean_y_test,  
    ↵ mitigation_types=[ "pruning", "filtering" ])
```

Neural Cleanse es una medida defensiva que implementa ART, pero otras defensas se pueden encontrar en el módulo [art.defences.transformer.poisoning](#).

6.6 Ataques de extracción

ART implementa ataques de extracción en el módulo [art.attacks.extraction](#). Todos los ataques son ataques de caja negra que emplean la técnica del modelo sustituto (ver Sección 2.1.3). Esta consiste en entrenar un modelo similar al modelo objetivo haciendo peticiones a este. ART implementa ataques como [Copycat CNN](#) o [Knockoff Nets](#), entre otros. El adversario debe definir un modelo que pueda imitar el modelo objetivo y debe crear una instancia del ataque que quiera llevar a cabo. Aquí, **victim** es el modelo objetivo y **classifier** es el modelo sustituto y **nb_stolen** es el número de peticiones al modelo objetivo.

```

from art.attacks.extraction import CopycatCNN, KnockoffNets

# Modelo sustituto
model = Sequential()
model.add(Conv2D(1, kernel_size=(7, 7), activation="relu",
    ↵ input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(4, 4)))
model.add(Flatten())
model.add(Dense(10, activation="softmax"))
model.compile(loss="categorical_crossentropy",
    ↵ optimizer=keras.optimizers.Adam(lr=0.001),
    ↵ metrics=[ "accuracy" ])

classifier = KerasClassifier(model=model, use_logits=False,
    ↵ clip_values=(0, 1))

# Ataque de extracción
attack = KnockoffNets(classifier=victim)
stolen_model = attack.extract(x=x_train, y=y_train,
    ↵ thieved_classifier=classifier, nb_epochs=10, nb_stolen=10000)

```

Este tipo de ataques no tienen sentido en modelos de caja blanca (conocemos todos los parámetros e hiperparámetros) como los que hemos visto hasta ahora, pero pueden servir para comprobar cómo de robustos son nuestros modelos frente a estos ataques. Por ello, nos centraremos en las defensas que proporciona ART para ataques de extracción.

ART implementa defensas a través del módulo `art.defences.postprocessor`, que se encargan de modificar la salida del modelo. Entre ellas, se encuentran el redondeo (fijar un número de decimales para las salidas del modelo) y el ruido gaussiano (que añade ruido a la salida del modelo). Aplicar este tipo de defensas a un modelo antes de entrenar es muy sencillo. Sólo hay que crear una instancia o instancias de las defensas que queremos aplicar e indicarlo en la clase `KerasClassifier` mediante el parámetro `postprocessing_defences`.

```

from art.defences.postprocessor import Rounded, GaussianNoise

# Defensas
postprocessor_rounded = Rounded(decimals=4)
postprocessor_gaussian = GaussianNoise(scale=0.1)

# Se añaden las defensas al modelo
victim_defense = KerasClassifier(model=model,
    ↵ clip_values=(min_pixel_value, max_pixel_value),
    ↵ use_logits=False,
    ↵ postprocessing_defences=[postprocessor_rounded,
    ↵ postprocessor_gaussian])
victim_defense.fit(x_train, y_train, batch_size=128, nb_epochs=10)

```

Como hemos visto a lo largo de este tutorial, ART permite **realizar tanto ataques como aplicar defensas** de los cuatro tipos de ataques que define el Adversarial Machine Learning. ART permite comprobar la robustez de nuestros modelos de *machine learning* frente a este tipo de ataques y protegerlos (si fuera necesario) antes de desplegarlos en producción. ART no es la única herramienta opensource (ver Tabla A.1) que permite hacer esto, pero sí que es **la más completa con respecto al número de ataques y defensas implementados**, cubriendo todos los tipos (extracción, inversión, envenenamiento y evasión), así como los tipos de datos (imágenes, audio, vídeos, ...). Su gran documentación y número de ejemplos permiten probar y testar nuestros modelos sin requerir gran esfuerzo. Como desventaja, algunos ataques/defensas no están disponibles para todos los *frameworks* o versiones del mismo, aunque es de esperar que esto se solucione en futuras versiones de la librería.

7 Futuro y conclusiones

Los modelos de *machine learning* son muy útiles en numerosas aplicaciones, pero **presentan vulnerabilidades** que pueden ser críticas en aplicaciones que requieran una especial seguridad (por ejemplo, los sistemas de conducción autónoma) y los que son entrenados con datos sensibles (por ejemplo, los datos clínicos). En el caso de los sistemas de conducción autónoma, es posible modificar una señal de forma imperceptible para que un adversario pueda cambiar el significado de la señal a su antojo. Si el modelo no es suficientemente robusto puede poner en riesgo grave a los ocupantes del vehículo. De forma similar, si entrenamos un modelo con datos sensibles, es posible que los datos de entrenamiento sean filtrados, produciendo una grave vulneración de la legislación en esta materia. Estos son sólo algunos de ejemplos de vulnerabilidades que pueden darse en este tipos de modelos. **Disponer de métodos que nos protejan de estas vulnerabilidades en un tema crítico en numerosos sectores y aplicaciones**. Pero no está todo perdido y existen medidas defensivas que nos permiten protegernos frente a todos los ataques descritos.

La securización de los modelos de *machine learning* siempre se ha dejado en un segundo plano, ignorando las potencias vulnerabilidades que pueden suponer estos. A raíz de ello nace el **Adversarial Machine Learning**, que es la rama intersección del *machine learning* y la seguridad informática que se encarga de evidenciar estas vulnerabilidades y **proponer métodos de defensa** frente a estos ataques, que se clasifican en cuatro tipos (extracción, inversión, envenenamiento y evasión). En este sentido, se ha desarrollado la [matriz de amenazas del Adversarial Machine Learning](#), que pretende posicionar estos ataques en un *framework* similar a [ATT&CK](#) del MITRE y con ello, ayudar a los analistas de seguridad frente a esta nueva amenaza.

Con respecto a las técnicas de extracción destacan los métodos de resolución de ecuaciones y el modelo sustituto. Los **escenarios más realistas requieren de entrenar un modelo sustituto**, que normalmente serán modelos de *deep learning*, ya que son capaces de aproximar otros tipos de modelos.

Pese a todas estas técnicas disponibles, no es sencillo extraer un modelo en un entorno real debido a que los hiperparámetros no son sencillos de obtener sin realizar una cantidad ingente de peticiones, incluso **siendo el robo de modelos equiparable al entrenamiento desde cero en cuanto a datos necesario y poder computacional** [15]. Esto hace que en muchas ocasiones sea inviable realizar este tipo de ataques a modelos reales. Aún así, existen defensas para proteger los modelos. Entre ellas, se incluyen evitar devolver un número elevado de decimales en los modelos, emplear técnicas de privacidad diferencial o emplear métodos específicos como PRADA o Adaptive Misinformation.

Los ataques de inversión permiten invertir el flujo de información de un modelo de *machine learning* permitiendo a un adversario **inferir cierto conocimiento sobre los datos de entrenamiento**. Los ataques se dividen en tres grupos: Membership Inference Attack, Property Inference Attack y reconstrucción. Los primeros son los más numerosos, los más estudiados en la literatura científica y, por tanto, los que presentan un mayor número de técnicas de ataque.

Un tema central es **entender por qué los modelos filtran información**. Se ha observado que el sobreentrenamiento, la complejidad de los datos, el número de parámetros y un número grande de clases, entre otras, hacen más propenso un modelo a estos ataques.

Los ataques de inversión se pueden realizar en entornos reales, aunque **numerosos ataques sólo son viables en determinados escenarios muy concretos**. Aún así, se han propuesto defensas para hacer frente a estos ataques. Entre las más destacadas se incluyen la **privacidad diferencial, que proporciona un compromiso entre la precisión del modelo y privacidad**. Otras defensas incluyen el uso de **ensembles, criptografía homomórfica, computación multipartite segura** y técnicas de regularización, entre otras.

Los ataques de envenenamiento permiten contaminar el conjunto de entrenamiento y producir resultados no deseados. Se pueden emplear para **modificar la frontera de decisión** y con ello predecir de forma incorrecta o **crear una puerta trasera en un modelo** de tal forma que no el modelo se comporte de forma normal en la mayoría de caso, pero dadas una ciertas entradas creadas por el adversario, permite a este producir resultados no deseados. Las BadNets fueron una de las primeras puertas traseras en redes neuronales, permitiendo ser irreconocibles al sólo modificar los pesos de un modelo original. Destacable, es que **se conserva la puerta trasera, aunque se vuelva a entrenar el modelo con otro conjunto de datos**. Este hecho abre a los repositorios de **modelos preentrenados como un vector de ataque**, ya que pueden contener puertas traseras y ser indetectables. Otras propuestas más modernas como TrojanNet permiten tener una tarea secreta y una pública en un mismo modelo, siendo la primera activada a través de una permutación secreta sólo conocida por el adversario.

Se han propuesto defensas para estos ataques, principalmente en dos aspectos: **la protección del dato y la protección del algoritmo**.

Por último, los ataques de evasión permiten a un adversario que un modelo clasifique de forma incorrecta. La forma de conseguir esto es **introducir pequeñas perturbaciones de ruido en la entrada**. Estas entradas maliciosas se llaman ejemplos adversarios y existen muchos métodos para conseguir evadir los modelos de *machine learning*. Los más numerosos se han aplicado en la clasificación de imágenes, aunque también existen para audio, texto, vídeo, grafos o la evasión de sistemas de detección de *malware*, entre otros.

En la literatura, se han propuesto numerosas defensas para proteger los modelos frente a estos ataques. Entre las defensas más destacadas, se incluye el entrenamiento adversario, aplicar transformaciones a los datos de entrada o esconder los gradientes, entre muchas otras. Estas últimas son de las más numerosas, pero cada vez que aparece una defensa nueva, en poco tiempo es conseguida evadir, aunque **los esfuerzos cada vez son mayores en conseguir defensas robustas y más difíciles de evadir**.

Actualmente, nos encontramos en un punto en el que aparecen defensas como nuevos ataques que evaden estas medidas, ya que no existe un consenso sobre cuáles son las medidas de evaluación que permitan a un modelo como seguro. Importante es mencionar que **los ataques del futuro serán adaptativos en distintos escenarios** y no sólo aplicable en un determinado entorno, ya que siempre existirán defensas que protejan frente a este ataque específico [14].

A modo de conclusión final, los despliegues de modelos de *machine learning* son cada vez más comunes y estos se pueden convertir en un nuevo vector de ataque por parte de adversarios, por lo que **conocer tanto las posibles vulnerabilidades como sus posibles remediaciones permiten y permitirán en el futuro proteger** (junto con el uso de herramientas opensource) a **una organización de manera efectiva frente a esta nueva amenaza**.

Referencias

- [1] Emiliano De Cristofaro. An overview of privacy in machine learning. *CoRR*, abs/2005.08679, 2020.
- [2] Vasisht Duddu, Debasis Samanta, D. Vijay Rao, and Valentina E. Balas. Stealing neural networks via timing side channels. *CoRR*, abs/1812.11720, 2018.
- [3] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security*, pages 1322–1333. ACM, 2015.
- [4] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 7:47230–47244, 2019.
- [5] Chuan Guo, Ruihan Wu, and Kilian Q. Weinberger. Trojannet: Embedding hidden trojan horse models in neural networks. *CoRR*, abs/2002.10078, 2020.
- [6] Yingzhe He, Guozhu Meng, Kai Chen, Xingbo Hu, and Jinwen He. Towards privacy and security of deep learning systems: A survey. *CoRR*, abs/1911.12562, 2019.
- [7] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N. Asokan. PRADA: protecting against DNN model stealing attacks. In *EuroS&P*, pages 512–527. IEEE, 2019.
- [8] Sanjay Kariyappa and Moinuddin K. Qureshi. Defending against model stealing attacks with adaptive misinformation. *CoRR*, abs/1911.07100, 2019.
- [9] Manish Kesarwani, Bhaskar Mukhoty, Vijay Arya, and Sameep Mehta. Model extraction warning in mlaas paradigm. In *ACSAC*, pages 371–380. ACM, 2018.
- [10] Maria-Irina Nicolae, Mathieu Sinn, Tran Ngoc Minh, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Ian M. Molloy, and Benjamin Edwards. Adversarial robustness toolbox v0.2.2. *CoRR*, abs/1807.01069, 2018.
- [11] Seong Joon Oh, Max Augustin, Mario Fritz, and Bernt Schiele. Towards reverse-engineering black-box neural networks. In *ICLR (Poster)*. OpenReview.net, 2018.
- [12] Shilin Qiu, Qihe Liu, Shijie Zhou, and Chunjiang Wu. Review of Artificial Intelligence Adversarial Attack and Defense Technologies. *Applied Sciences*, 9(5):909, mar 2019.

- [13] Maria Rigaki and Sebastian Garcia. A survey of privacy attacks in machine learning. *CoRR*, abs/2007.07646, 2020.
- [14] Florian Tramèr, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. *CoRR*, abs/2002.08347, 2020.
- [15] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618. USENIX Association, 2016.
- [16] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *IEEE Symposium on Security and Privacy*, pages 36–52. IEEE Computer Society, 2018.
- [17] Yi Shi, Y. Sagduyu, and A. Grushin. How to steal a machine learning classifier with deep learning. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–5, 2017.

A Resumen de herramientas opensource

Tabla A.1: Recopilación de las principales herramientas opensource de Adversarial Machine Learning.

Nombre	Tipo	Algoritmos	Tipo de ataque	Ataque/Defensa	Frameworks soportados	Doc.	¿Activo?	Popularidad	Desarrollador
Athena 	Imagen	Deep Learning	Evasión	Defensa	Tensorflow JAX Torch	—	✗	✓	22 AlSysLab
Cleverhans 	Imagen	Deep Learning	Evasión	Ataque	Tensorflow Keras PyTorch Theano MxNet Lasagne	✓	✓	4800 Tensorflow	BethgeLab
Foolbox 	Imagen	Deep Learning	Evasión	Ataque	Tensorflow Keras PyTorch Theano MxNet Lasagne	✓	✓	1700	BethgeLab
ART 	Cualquiera	Deep Learning SVM Regresión logística	Evasión Extracción Inversión Envenenamiento	Ambos	Tensorflow Keras PyTorch Scikit Learn	✓	✓	1800	IBM
SECMIL 	Imagen	Scikit-Learn ... Deep Learning	Evasión Envenenamiento	Ataque	Scikit Learn PyTorch	✓	✓	20	SecML
Artificial Adversary 	Texto	—	Evasión	Ataque	—	✗	✗	335	Airbnb
DEEPSEC 	Imagen	Deep Learning	Evasión	Ambos	PyTorch	✗	✗	125	Xiang Ling
Adversarial Audio Examples 	Audio	DeepSpeech	Evasión	Ataque	—	✗	✗	170	Nicholas Carlini
Adversarial 	Imagen	Deep Learning	Evasión Evasión	Ambos Ambos	PyTorch Tensorflow PaddlePaddle	—	✓	730	BorealisAI
Armory 	Imagen	Deep Learning	Envenenamiento	—	PyTorch Tensorflow PaddlePaddle	✓	✓	395	Two Six Labs
AdvBox 	Imagen	Deep Learning	Evasión	Ambos	PyTorch Caffe2 MxNet Keras	✓	✓	990	Baidu
DeepRobust 	Imagen Grafos	Deep Learning	Evasión	Ambos	Tensorflow Pytorch	✗	✓	275	DataScience and Engineering Lab (MSU)

Acerca de los autores

Miguel Hernández Boza

miguel.hernandez2.next@bbva.com

Security Researcher



Estudiante de por vida, apasionado por I+D. Nacido en Zaragoza, ingeniero de teleco por la Universidad de Zaragoza y máster en ciberseguridad por la Universidad Carlos III. Actualmente trabaja en el laboratorio de investigación en seguridad para BBVA Next Technologies como Security Researcher, generando nuevas soluciones innovadoras de seguridad. Ponente en diversas charlas de seguridad nacionales como Rooted-Con, NavajaNegra, NoConName, Codemotion o Cybercamp, internacionales como HITB y otras académicas como JNIC o Secrypt. Colaborador en centros docentes con el programa #cibercooperantes. Proyectos open source: GrafScan y DeepConfusables.

José Ignacio Escribano Pablos

joseignacio.escribano.pablos.next@bbva.com

Cybersecurity and Machine Learning Researcher



Actualmente está realizando su tesis doctoral sobre criptografía post cuántica. Es graduado en Matemáticas e Ingeniería del Software y máster en Ingeniería de la Decisión. Sus principales intereses son la criptografía y el adversarial Machine Learning. Además, es contribuidor principal de algunos proyectos Open Source como ASSAP o Deep Confusables. Ponente de Cybercamp y Hack In The Box.

Acerca de Labs

Exploramos el potencial de nuevas tecnologías, de modelos de interacción entre personas y máquinas, y de nuevos enfoques y técnicas de seguridad para fomentar la evolución tecnológica de BBVA Next Technologies y de nuestros clientes. Encontramos respuestas a preguntas que aún no han sido formuladas. Nuestra misión es dar un impulso al ecosistema tecnológico, posicionándonos como referentes del mercado en nuestras líneas estratégicas. ¿Qué nos define? Una actitud colaborativa. Trabajamos de manera cercana con universidades, partners tecnológicos y otras entidades de la industria que nos permiten ir un paso por delante. ¿Qué nos mueve? Transferir el conocimiento a cada área tecnológica de los equipos para ser capaces de traducir esa ambición en realidad.

Acerca de BBVA Next Technologies

Somos mentes inquietas, curiosas, a las que nos apasiona lo que hacemos: buscar soluciones tecnológicas inteligentes y resolver retos que aún no han sido planteados. Más de 1.050 personas entre España y México que forman parte de una compañía del Grupo BBVA experta en ingeniería de software, creada para acelerar su transformación digital y impulsar a otras empresas líderes. En un mundo que evoluciona sin parar, hemos elegido ser parte del cambio. Nuestro modelo tecnológico y de talento es único; ponemos el foco en la excelencia tecnológica a través de la divulgación del conocimiento y el networking colaborativo. Fomentamos el orgullo técnico con una cultura flexible capaz de adaptarse al ritmo de la evolución constante, lo que nos permite ser pioneros dentro del sector e ir un paso por delante. #PeopleFirst es nuestro primer valor como compañía. Lo primero son las personas y por eso impulsamos el desarrollo profesional potenciando el talento desde dentro a través de un plan de carrera personalizado, aprendizaje constante y formación continua. Creemos que la fuerza de cada proyecto está en todos los que trabajamos en #BBVANextTechnologies; personas comprometidas e inconformistas a las que nos mueve pensar más allá de lo convencional y cuestionar lo establecido. Estamos orgullosos de nuestro ADN que refleja nuestro espíritu libre y creativo, nuestra pasión por la tecnología, por crear proyectos de alto impacto y de manera diferencial, pero sobre todo, estamos orgullosos de poder hacerlo juntos, como un equipo.

BBVA
Next Technologies