Universidad de la República
Facultad de Ingeniería

Proyecto de Grado

Ataques adversarios en sistemas de reconocimiento visual

Diego Irigaray
Camila Serena

Tutores:
Dr. Mauricio Delbracio
Dr. José Lezama
Dr. Guillermo Moncecchi

29 de noviembre de 2019

Resumen

En la actualidad, las redes neuronales alcanzan el estado del arte en la mayoría de tareas que emplean técnicas de aprendizaje automático, siendo cada vez más los problemas del mundo real en los cuales se hace uso de este tipo de modelos. Sin embargo, se ha demostrado recientemente que dichas redes son vulnerables a los denominados ejemplos adversarios: datos de entrada que tras ser levemente modificados logran engañar a las redes haciéndolas devolver resultados incorrectos.

En este trabajo realizamos un estudio sobre el fenómeno de los ejemplos adversarios, centrándonos para ello en los sistemas de visión artificial. Estudiamos cómo se obtienen dichos ejemplos, qué características presentan y por qué existen, así como algunas de las principales propuestas que buscan mitigar esta vulnerabilidad.

Adicionalmente implementamos un *framework* orientado al desarrollo y evaluación de algoritmos tanto de ataque como de defensa contra ejemplos adversarios, el cual utilizamos para realizar distintos experimentos sobre algunos de los métodos más relevantes propuestos hasta el momento.

Palabras Clave — Ataques adversarios, Ejemplos adversarios, Redes neuronales, Visión artificial

Índice general

Capítulo 1

Introducción

En los últimos años se han hecho numerosos avances en el área del aprendizaje automático. Esto se debe en parte a la gran cantidad de datos disponibles en la actualidad, necesarios para el entrenamiento de los modelos, así como a las importantes mejoras realizadas en el *hardware* que hacen posible contar con un poder de cómputo cada vez mayor. Gracias a estos avances hoy en día las redes neuronales profundas son capaces de obtener excelentes resultados en múltiples áreas, tales como visión artificial [1], reconocimiento de voz [2] y procesamiento del lenguaje natural [3], entre otras. No es de extrañar por lo tanto que estas técnicas sean cada vez más aplicadas en distintos problemas del mundo real, desde vehículos autónomos, traductores automáticos y asistentes inteligentes a aplicaciones en la medicina, videojuegos y publicidad.

A pesar del buen desempeño alcanzado por las redes neuronales, se ha descubierto que son altamente sensibles a los denominados **ejemplos adversarios**[4], datos de entrada ligeramente alterados con el objetivo de obtener resultados incorrectos por parte de la red. Para generar estos ejemplos basta con que el atacante realice ciertas modificaciones específicas sobre los datos reales, las cuales pueden incluso resultar imperceptibles para un humano pero que aún así son capaces de afectar significativamente la salida generada. Por ejemplo, para el caso de las redes de clasificación de imágenes estos consisten en ejemplos levemente modificados, que a pesar de verse básicamente iguales a las imágenes originales son clasificados incorrectamente.

Es claro que la robustez es una característica sumamente importante en los modelos de aprendizaje automático, por lo que la existencia de estos ejemplos adversarios propone una seria amenaza a muchas de sus aplicaciones. Entre otros escenarios críticos tenemos a los vehículos autónomos y a los sistemas de reconocimiento facial, sobre los cuales un atacante podría alterar la percepción del vehículo o hacerse pasar por una persona diferente.

Por estos motivos los ataques adversarios han sido un activo tema de investigación desde sus primeras menciones en el año 2014 [4]. Se han propuesto numerosos algoritmos para generarlos así como mecanismos para defender a las redes de dichos ataques, lográndose grandes avances no solo en lo relacionado a la robustez de estos sistemas sino que también en cuanto a la comprensión que se tiene sobre las redes neuronales en general. De todas formas, hasta esta fecha no se han logrado desarrollar redes invulnerables a este fenómeno, permaneciendo abierta la pregunta de si realmente es posible eliminar dicha vulnerabilidad o de si es una característica inherente a este tipo de modelos.

Para promover la investigación en esta área, en el año 2017 se presentó la primer competencia oficial en el marco del evento *Neural Information Processing Systems* (NeurIPS), conferencia de aprendizaje automático y neurociencia considerada de las mas importantes

en su tipo, en la cual se recibieron algoritmos de ataques y defensas para ser evaluados entre si. La competencia volvió a repetirse al año siguiente recibiendo más de 3000 propuestas de algoritmos, lo que de cierta manera refleja el impacto que ha tenido este tema entre los investigadores.

Si bien es posible encontrar ejemplos adversarios para múltiples sistemas basados en aprendizaje automático, en este proyecto nos centramos únicamente en los ataques y defensas sobre redes de clasificación de imágenes.

1.1. Ataques Adversarios

Se les suele llamar **Ataques Adversarios** a los distintos algoritmos diseñados para generar ejemplos adversarios. Estos últimos fueron estudiados originalmente sobre las redes de clasificación de imágenes donde se notó que era posible para un atacante generar imágenes perturbadas casi indistinguibles de las originales y que sin embargo eran clasificadas incorrectamente por la red [4]. Esto resulta contraintuitivo debido a la gran capacidad de generalización que presentan las redes neuronales, pero tal vez más curioso es que estos ejemplos adversarios no solo afectan a la red para la cual fueron generados sino que también logran en muchos casos engañar a otras redes, entrenadas con distintos hiper-parámetros e incluso sobre conjuntos de datos disjuntos [4].

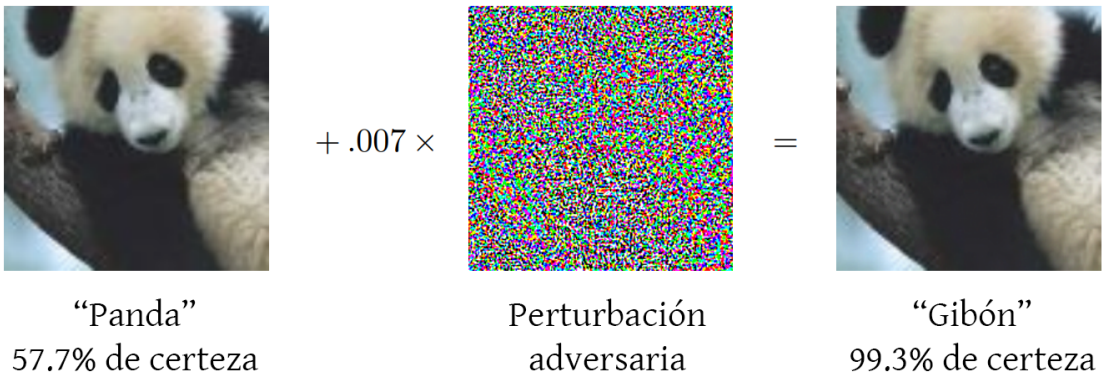


Figura 1.1: Ejemplo adversario generado por el ataque FGSM, extraída de [5]

La mayoría de los ataques hacen uso de una de las propiedades intrínsecas de las redes neuronales, su diferenciabilidad, y generan ejemplos adversarios abordándolo como un problema de optimización. En estos se parte de un conjunto de datos de entrada "limpios", los cuales pueden interpretarse como puntos en un espacio multidimensional, y los desplazan en la dirección que maximice el error de la red, normalmente sujeto a alguna restricción de distancia entre los datos perturbados y los originales. De esta forma se logran generar ejemplos capaces de engañar a las redes y que al mismo tiempo se asemejan a los datos naturales.

Posteriores estudios demostraron que no solo se obtienen ejemplos adversarios al modificar directamente la entrada de las redes, sino que también es posible trasladar estos ataques al mundo físico [6, 7, 8]. Esto refiere a los casos en los cuales la perturbación es primero llevada al mundo físico, por ejemplo siendo impresa, y posteriormente proporcionada a la red mediante una cámara u otro sensor, encontrándose sujeta a distorsiones y diversos factores externos. Los autores fueron capaces de engañar a los distintos modelos evaluados

utilizando ejemplos adversarios impresos y luego fotografiados, a pesar del ruido introducido en ambas etapas. Esto eleva aún más el riesgo propuesto por los ataques adversarios debido a que extiende los escenarios en los que estos pueden ser utilizados, como en este caso mediante perturbaciones directamente aplicadas sobre objetos en el mundo real.

Otras observaciones realizadas, también en el contexto de redes de clasificación de imágenes, indican que es posible encontrar pequeñas perturbaciones que al ser agregadas a una imagen cualquiera logran engañar al clasificador con alta probabilidad [9]. Estas perturbaciones no solo son universales en el sentido de generar ejemplos adversarios independientemente de la imagen subyacente sino que, según lo reportado, también generan ejemplos adversarios para redes con distinta arquitectura, sumándole aún otra vulnerabilidad a estos sistemas.

1.2. Objetivos y Contribuciones

Creemos que los ataques adversarios son un tema de suma importancia dado que atentan contra la seguridad de las redes neuronales, una de las áreas más relevantes de la computación en la actualidad. Por esto, nuestro principal objetivo es el de estudiar el problema de los ejemplos adversarios, las características que presentan y algunos de los ataques y defensas más relevantes, centrándonos para esto en las redes de clasificación de imágenes.

Otro objetivo consiste en la realización de experimentos propios para complementar dicho estudio. Estos experimentos se ven afectados por una gran cantidad de factores, como el *dataset* utilizado, arquitectura de la red, parámetros de entrenamiento, etc., los cuales hacen difícil reproducir y comparar resultados. Teniendo en cuenta esto nos proponemos también a desarrollar un *framework* que sirva como una herramienta para experimentar con ataques y defensas, la cual pueda ser de utilidad no solo para nosotros sino también para otros investigadores.

Los objetivos del proyecto por lo tanto incluyen:

- Estudiar el problema de los ejemplos adversarios.
- Analizar algoritmos de ataques y defensas destacados.
- Reproducir los resultados obtenidos por algunos autores.
- Desarrollar un *framework* para la evaluación de ataques y defensas.

1.3. Estructura del Informe

El segundo capítulo contiene una base teórica previa al estudio de los ejemplos adversarios, comenzando por algunos conceptos generales como el de aprendizaje automático y redes neuronales, para luego avanzar a temas más específicos a nuestro caso de estudio como lo son las redes convolucionales y el problema de clasificación.

En el tercer capítulo presentamos el estudio realizado sobre los ejemplos adversarios. Comenzamos por una descripción general del problema, repasamos las principales propiedades que presentan estos ejemplos e introducimos algunos de los algoritmos de ataque más relevantes. Finalmente realizamos un estudio de las defensas utilizadas contra estos ataques, nuevamente comentando sus características y algunas de las estrategias más conocidas.

El cuarto capítulo consiste en la descripción del *framework* desarrollado. En este describimos los distintos componentes involucrados, con sus respectivas características y funcionalidades, así como el funcionamiento global del *framework*.

En el quinto capítulo presentamos los resultados obtenidos mediante los experimentos realizados. Estos consisten en la evaluación de varios ataques y defensas de los presentados en el capítulo 3, donde comparamos los resultados obtenidos con los publicados por los respectivos autores.

Finalmente en el capítulo seis incluimos las conclusiones obtenidas y posible trabajo a futuro.

Capítulo 2

Marco Teórico

Si bien el foco central de este proyecto representa una pequeña parte dentro de un área tan amplia y variada como lo es el aprendizaje automático, creemos que una base de determinados temas resulta de suma utilidad al momento de estudiar en que consisten y cómo funcionan los ejemplos adversarios. Por eso en las próximas secciones presentamos de forma breve una serie de conceptos relevantes, comenzando por los más generales y avanzando hacia los más específicos, para luego en el capítulo 3 abordar el tema central de esta informe.

2.1. Aprendizaje Automático

El aprendizaje automático o *machine learning* es una de las ramas más populares de la inteligencia artificial. La primera aparición del término *machine learning* se da en el año 1959 por Arthur Samuel, quien lo define como el área de estudio que le da a las computadoras la habilidad de aprender sin ser explícitamente programadas.

Tom Mitchell [10] provee luego una definición más formal:

> Se dice que un programa aprende de la experiencia E con respecto a la tarea T y con cierta medida de desempeño P, si su desempeño en la tarea T, medido por P, mejora con la experiencia E.

En otras palabras, el aprendizaje automático se encarga de construir programas capaces de mejorar su desempeño en cierta tarea a partir de la experiencia. La idea detrás de estos algoritmos es que tras observar grandes cantidades de datos el sistema sea capaz de extraer y aprender información relevante que posteriormente sea de utilidad, por ejemplo, para realizar predicciones sobre datos nuevos.

Volviendo a la definición de Tom Mitchell, detallamos brevemente los tres componentes involucrados.

1. **Experiencia**: Los datos son un importante pilar de cualquier algoritmo de aprendizaje automático y en general hacen falta grandes cantidades de ellos para obtener buenos resultados (del orden de miles de ejemplos o más). Estos datos se organizan en los llamados *datasets*, que básicamente son conjuntos de muchos ejemplos del problema a abordar. Cada uno de estos ejemplos a su vez suele representarse como un vector $x \in \mathbb{R}^n$ compuesto de elementos denominados *features*.

Muchos de los algoritmos de aprendizaje automático pueden ser clasificados dentro de dos categorías según el tipo de experiencia usada durante el entrenamiento: aprendizaje **supervisado** y aprendizaje **no supervisado**.

- En el aprendizaje supervisado, los ejemplos del *dataset* cuentan además con una etiqueta o valor objetivo asociado y se intenta que el programa aprenda a predecir este valor para nuevos ejemplos de entrada. Se le llama supervisado debido a que depende de un supervisor u oráculo que ingrese los valores correctos para cada ejemplo, indicándole al sistema cómo se debe comportar.

  Dentro de esta categoría se encuentran los problemas de clasificación en los que nos centramos. Estos consisten en predecir las categorías correctas para cada entrada, eligiéndolas a partir de un conjunto discreto de opciones.

- Por otro lado en el aprendizaje no supervisado el algoritmo cuenta únicamente con los ejemplos de entrada, sin información agregada. En estos casos se pretende que mediante el uso de dichos datos el programa sea capaz de aprender ciertas propiedades de interés sobre la estructura del *dataset*.

En el resto del informe tratamos exclusivamente con algoritmos de aprendizaje supervisado, debido a que son los utilizados para resolver los problemas de clasificación en los cuales nos estaremos centrando.

2. **Tarea**: Esta refiere a la tarea que se espera que el programa sea capaz de llevar a cabo una vez finalizada la etapa de entrenamiento. Suelen tratarse de tareas bastante difíciles de resolver mediante algoritmos tradicionales programados directamente por humanos. Hoy en día existe una gran variedad de tareas en las cuales se obtienen muy buenos resultados mediante el uso de aprendizaje automático, como son la clasificación y segmentación de imágenes, traducción automática, detección de anomalías y muchas más.

3. **Desempeño**: Finalmente hace falta una forma cuantitativa de determinar qué tan bien se comporta el programa. Para esto se utiliza cierta medida de desempeño, la cual en general dependerá de la tarea llevada a cabo por el programa.

   Una medida de desempeño muy usada para problemas de clasificación es la **precisión** o **accuracy**. Esta refiere a la proporción de ejemplos del *dataset* sobre la cual se obtuvo la salida correcta.

Una vez obtenido el *dataset* y definidas la tarea y la medida de desempeño, aún no queda del todo claro cómo logran aprender los programas. Para esto se hace uso de los llamados **modelos**, siendo las **redes neuronales** uno de los tipos más populares en la actualidad. De forma simplificada, las redes neuronales son funciones de la forma $f(\theta, x) = y$, que hacen uso de conjuntos de **parámetros** ó **pesos** θ y los *features* de la entrada x para generar una salida determinada.

Luego, lo que inicialmente definimos como aprender se resume a encontrar los valores adecuados para estos parámetros, es decir, los que hacen que la red obtenga el mejor resultado posible. Esto se realiza durante un proceso conocido como **entrenamiento** y consiste en resolver un problema de optimización en el cual se hacen uso de otros dos componentes: la **función de costo** o **función de pérdida** y el **mecanismo de optimización**.

La función de costo depende del problema tratado y se define de tal forma que minimizar este costo indirectamente mejore el desempeño de la red. Por ejemplo, en los problemas de clasificación se utiliza una función de costo que retorna valores cercanos a cero si se predicen las clases correctas, y valores más altos si se predicen clases equivocadas.

El mecanismo de optimización por otro lado suele ser el mismo en la mayoría de los casos, **descenso por gradiente estocástico** o alguna de sus variantes. Esto implica evaluar iterativamente la función de costo para conjuntos reducidos de ejemplos del *dataset* actualizando el valor de los parámetros en la dirección opuesta al gradiente.

Si bien dicha optimización se realiza sobre los datos de entrenamiento, el verdadero objetivo de estos programas es el de desempeñarse bien sobre ejemplos que no hayan sido procesados previamente. A esto se conoce como la capacidad de **generalizar** de los modelos.

En la práctica, para determinar qué tan bien se comporta un algoritmo sobre datos nuevos se suele dividir el *dataset* utilizado en dos subconjuntos disjuntos. De esa forma se utiliza uno de ellos durante la etapa de entrenamiento y el otro para evaluar el desempeño sobre datos desconocidos. A los errores obtenidos durante el entrenamiento y la evaluación se les denomina **error de entrenamiento** y **error de evaluación** respectivamente.

Esto da lugar a dos problemas importantes del aprendizaje automático, denominados **underfitting** y **overfitting**.

- El *underfitting* corresponde a un error de entrenamiento y evaluación demasiado alto. Esto puede deberse a que el modelo no es lo suficiente expresivo como para aprender una función que logre buenos resultados sobre los datos tratados.
- El *overfitting* sucede al obtener una diferencia demasiado grande entre el error de entrenamiento y el error de evaluación. Esto puede significar que en este caso el modelo es demasiado expresivo, y logra aprender propiedades específicas a los ejemplos de entrenamiento que no se generalizan bien a nuevos datos.

El objetivo de cualquier algoritmo de aprendizaje automático por lo tanto, es hacer que tanto el error de entrenamiento como su diferencia con el error de evaluación sean lo más pequeños posible.

2.2. Redes Neuronales Feedfoward

Las redes neuronales *feedfoward* constituyen una de las familias de modelos más utilizadas en el aprendizaje automático, debido principalmente a que logran obtener excelentes resultados en numerosas tareas.

El objetivo de estas redes es el de aproximar alguna función determinada del tipo $y = f^*(x)$, en general compleja. Para esto define una correspondencia $y = f(x, \theta)$ en el cual θ es un conjunto de parámetros (los pesos de la red) los cuales deben ser aprendidos para obtener la mejor aproximación de f^* posible.

Reciben el nombre de redes debido a que generalmente consisten en una composición de funciones de la forma $f = f^{(n)}(...(f^{(2)}(f^{(1)}(x))))$. A cada una de estas funciones se las conoce también como **capas** de la red, donde la última de ellas (f^n) recibe el nombre de **capa de salida** y las restantes el de **capas ocultas**, refiriéndose además como **profundidad** de la red a su número total de capas.

Estas redes se suelen representar mediante grafos acíclicos y dirigidos como el de la figura 2.1. En esta se puede observar que la información fluye a través de toda la red, comenzando por la entrada y siguiendo a través de todas las capas ocultas hasta finalmente generar el resultado de la red, motivo por el cual se les llama prealimentadas.

Finalmente el nombre neuronales se debe a que están ligeramente inspiradas en el sistema nervioso animal. Cada capa oculta aplica una función vectorial $f^{(j)} : \mathbb{R}^{n_j} \to \mathbb{R}^{n_{j+1}}$ que

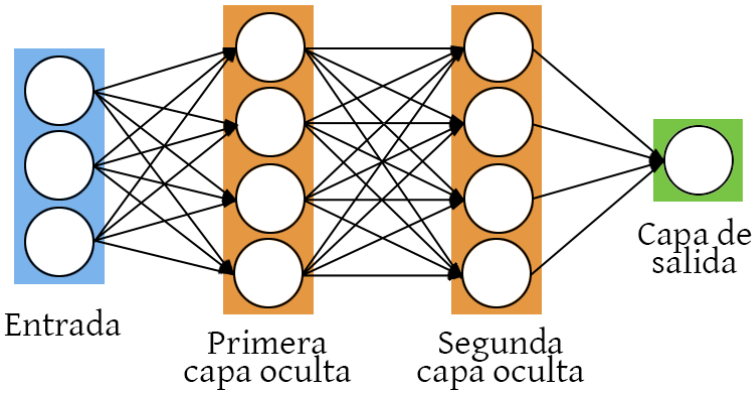


Figura 2.1: Diagrama de una red neuronal prealimentada de dos capas ocultas.

a su vez puede interpretarse como una serie de funciones $[f^{(j)}]_i$ que reciben un vector y devuelven un valor unidimensional denominado **activación**. El comportamiento de cada una de estas sub-funciones es análogo al de una neurona en el sentido de que reciben su entrada desde un grupo de neuronas y produce su propia salida, la cual eventualmente se propaga a otras neuronas de la red. Si cada neurona utiliza la salida completa de la capa anterior, las capas reciben el nombre de **totalmente conectadas**.

Durante el entrenamiento contamos con un conjunto de puntos (x_i, y_i) donde cada y_i es una aproximación de f^* evaluada en el punto x_i. Esto nos indica que para una entrada x_i, el resultado de la capa de salida debe ser similar a y_i, pero no especifica nada acerca del comportamiento de las capas ocultas, siendo responsabilidad del algoritmo de aprendizaje el determinar cómo utilizarlas para obtener una buena aproximación de f^*.

2.3. Redes Convolucionales

Las redes convolucionales son un tipo especial de redes neuronales. Estas comparten muchas características con las redes tradicionales presentadas en la sección anterior pero se diferencian en que están optimizadas para procesar datos con una estructura de grilla similar al de las imágenes.

Recordando lo mencionado 2.1, cada capa de las redes tradicionales está compuesta por un conjunto de neuronas las cuales interactúa con todas las salidas de la capa anterior de forma independiente al resto de neuronas de la capa. Esto hace que la cantidad de parámetros necesaria para trabajar con imágenes (incluso de tamaños relativamente pequeños) crezca rápidamente al aumentar la profundidad de la red, incrementando sus requerimientos de memoria y volviéndola más propensa a *overfitting*. Para evitar ambos problemas, las redes convolucionales hacen uso de las denominadas **capas convolucionales**, que como explicamos en la próxima sección sacan provecho de la estructura de las imágenes para reducir la cantidad de parámetros necesarios. En conjunto a estas también es común usar las llamadas capas de *pooling* y las ya mencionadas capas totalmente conectadas. Las de *pooling* básicamente ayudan a reducir aún más la cantidad de parámetros mientras que las totalmente conectadas se agregan al final de la red para generar una salida con las dimensiones deseadas.

A continuación describimos en qué consisten las capas convolucionales y cómo ayudan a procesar imágenes de forma más eficiente.

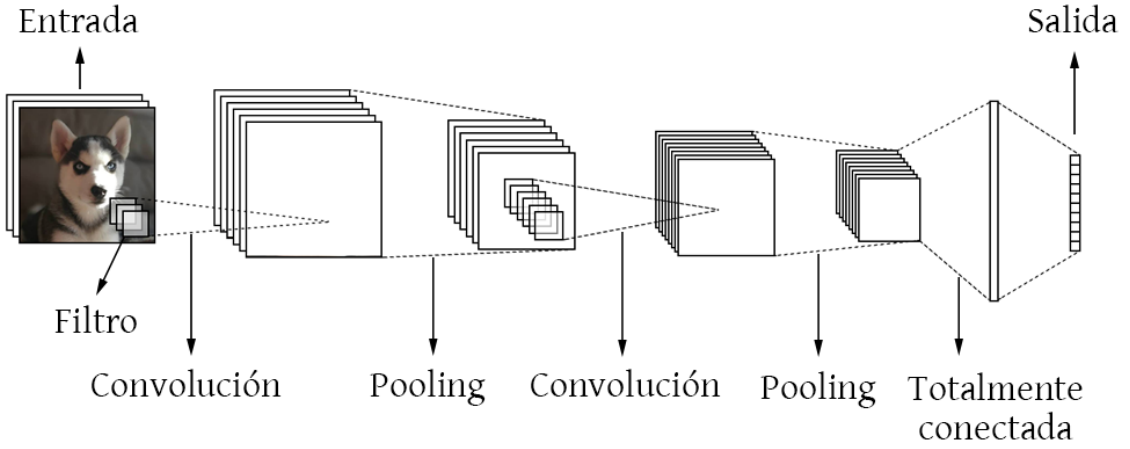


Figura 2.2: Representación de una red convolucional, con dos capas de convolución, dos capas de *pooling* y una capa totalmente conectada.

2.3.1. Capa convolucional

Las capas de convolución son el componente más importante en este tipo de redes. A diferencia de las capas totalmente conectadas, estas organizan las neuronas en una estructura tridimensional mediante la cual se encargan de transformar un **volumen de activaciones** en otro. Por ejemplo, la primera capa oculta puede recibir como entrada un volumen de dimensiones $32 \times 32 \times 3$ (correspondiente a una imagen a color) y retornar otro volumen de dimensiones $32 \times 32 \times 12$, aumentando en este caso su profundidad.

Otra diferencia es que cada neurona de estas capas se conecta solamente a una pequeña región del volumen de entrada, aunque siempre con la totalidad de la profundidad. Siguiendo el ejemplo anterior, una neurona de la primera capa podría conectarse por ejemplo a un volumen de tamaño $5 \times 5 \times 3$.

Estas neuronas cuentan con conjuntos de parámetros los cuales son utilizados para calcular el producto interno con las regiones correspondientes de la entrada, retornando un único valor por cada una. De esta forma, entre todas las neuronas de un mismo nivel de profundidad se genera una salida bidimensional denominada **mapa de activación**.

Finalmente, para poder reducir la cantidad de parámetros de estas capas se tiene una consideración adicional. Esta consiste en asumir que si los parámetros de una neurona le permiten extraer información de utilidad en determinada región de la entrada (como por ejemplo ciertas líneas o curvas de una imagen), entonces también sería de utilidad extraer esa información en otras regiones diferentes. Con esto en mente se agrega la restricción de que todas las neuronas en un mismo nivel de profundidad utilicen un único conjunto de parámetros compartidos.

Al agregar esta restricción, el procesamiento realizado por cada capa puede verse como una **convolución** de matrices en la cual los pesos de sus neuronas se "desplazan" a lo largo y ancho de la imagen de entrada para producir los mapas de activación[1], reduciendo significativamente la cantidad de parámetros necesarios. Debido a esto reciben el nombre de capas y redes convolucionales, y a los parámetros de estas neuronas se les refiere como

[1]En [11] se pueden observar varios ejemplos gráficos de la operación de convolución aplicada por las redes.

filtros o **núcleos**.

Además del ancho y alto de los filtros (recordar que la profundidad debe ser igual a la profundidad de la entrada), las capas convolucionales dependen de otros hiperparámetros que determinan las dimensiones del volumen de salida. Estos son:

- **Profundidad**: Este hace referencia al número de filtros utilizados en la capa (no confundir con la profundidad de la red), lo que indirectamente determina la profundidad que tendrá el volumen de salida.
- **Stride**: El *stride* refiere a la separación con la que se desplazan los filtros a través de la imagen. Valores mayores a uno resultan en una salida más pequeña que la entrada (en ancho y alto).
- **Zero-padding**: Cantidad de "anillos" de ceros agregados alrededor del volumen de entrada. En general se utilizan para evitar reducir el tamaño de la salida con respecto al de la entrada.

Al reducir la cantidad de parámetros necesarios por cada capa se hace posible aumentar de forma considerable la profundidad de estas redes, lo que en la práctica se traduce a un mejor desempeño sobre las distintas tareas realizadas. En estos escenarios las redes logran aprender los conceptos en una forma jerárquica mediante múltiples niveles de abstracción, comenzando por las primeras capas las cuales aprenden a reconocer conceptos simples como por ejemplo líneas y colores, hasta las capas más profundas que empiezan a reconocer formas y finalmente objetos.

2.4. Clasificación

Como mencionamos anteriormente, nos estaremos centrando en redes de clasificación, puntualmente clasificación de imágenes. En este contexto nos referimos a clasificación como la tarea de asignar una clase o categoría discreta a los distintos datos de entrada. Formalmente, se desea que el algoritmo logre reproducir cierta función de la forma $f : \mathbb{R}^n \to \{1, 2, ..., k\}$.

En la mayoría de las redes de clasificación estudiadas se utiliza una variación de esta función en la cual la salida es un vector de probabilidad de las clases. Este es un vector con tantos elementos como categorías reconocidas en el cual el valor de cada elemento se corresponde a la probabilidad asignada por la red a la clase correspondiente.

Para el caso de clasificación de imágenes, la entrada consiste en una matriz (posiblemente multidimensional) donde cada elemento representa el valor de un píxel (o un canal) y la salida es una predicción del objeto presente en la imagen. Si bien esto es una tarea trivial para los humanos, claramente presenta una serie de dificultades para una computadora, que debe ser capaz de procesar esta matriz de números e identificar los objetos presentes, incluso en distintos escenarios, con distintos tamaños, puntos de vista, iluminación, etc.

Resolver este problema es una tarea de suma importancia debido a que no solo tiene utilidad por sí misma, sino que también es utilizada para resolver problemas más complejos como detección de objetos y segmentación [12].

En el entrenamiento de modelos de clasificación, se busca ajustar los parámetros de la red de forma tal que se minimice una función de ajuste o perdida, que penalice cuando la probabilidad estimada por el modelo para una entrada difiera de su clase correcta. La función comúnmente utilizada en estos casos es *cross-entropy* o discrepancia de entropía

cruzada. Sea $\mathbf{p_i}$ el vector de probabilidades dado por la red para la entrada $\mathbf{x_i}$ y $\mathbf{y_i}$ el vector *one-hot encoded*[2], donde la clase correcta es y_i

$$L = \sum L_i \text{ con } L_i = -\log[p_i]_{y_i}$$

La medida de desempeño más utilizada en los modelos de clasificación, es la precisión o *accuracy* de la red, es decir, el porcentaje de ejemplos que son clasificados correctamente. Adicionalmente, existen casos donde debido a la gran cantidad de clases reconocidas, muchas de las cuales resultan similares entre sí, se puede llegar a considerar una predicción como correcta si el valor asignado a la clase real se encuentra dentro de las n clases con mayores probabilidades. Luego es normal reportar la precisión de las redes en función de los denominados *top-1* y *top-5 accuracy*, refiriéndose a la probabilidad de que la clase correcta sea la primera predicción o se encuentre entre las primeras cinco predicciones respectivamente.

2.5. Datasets

Como vimos en la sección 2.1, se utilizan los denominados *datasets* tanto para entrenar como para medir el desempeño de los distintos modelos. A continuación presentamos algunos de los *datasets* más populares utilizados para la clasificación de imágenes y con los que trabajamos durante este proyecto. En la figura 2.3 se pueden observar imágenes de ejemplo para cada uno de estos.

2.5.1. MNIST

El MNIST es una base de datos de dígitos escritos a mano. Consta de un total de 70.000 ejemplos, 60.000 de entrenamiento y 10.000 de evaluación, de imágenes en blanco y negro con los dígitos desde el 0 al 9 y un tamaño de 28×28 píxeles. Es uno de los *datasets* más simples utilizados en la clasificación de imágenes, en el cual las redes modernas obtienen fácilmente un *accuracy* de casi el 100 %.

2.5.2. CIFAR-10

Este es un *dataset* de imágenes a color. Contiene un total de 60.000 imágenes de tamaño 32×32 píxeles, de las cuales 50.000 pertenecen al conjunto de entrenamiento y 10.000 al de evaluación. Al igual que en MNIST, todas las imágenes de este *dataset* pertenecen a una de diez clases posibles, en este caso de distintos animales y vehículos como por ejemplo perro, rana, avión, etc.

2.5.3. ImageNet

ImageNet es una base de datos de imágenes organizada siguiendo la jerarquía de *WordNet* [3], actualmente sólo para sustantivos. Contiene alrededor de 14 millones de imágenes pertenecientes a más de 200.000 clases, aunque se suele usar una versión reducida del *dataset*

[2] Vector de dimensión igual a la cantidad de clases posibles, con un uno representando la clase correcta y ceros para las demás categorías.

[3] https://wordnet.princeton.edu/

que lo limita a 1.000 clases disjuntas. Sus imágenes varían en tamaño pero son considerablemente más grandes que la de los otros *datasets* mencionados, en general de tamaños superiores a 256×256 píxeles.

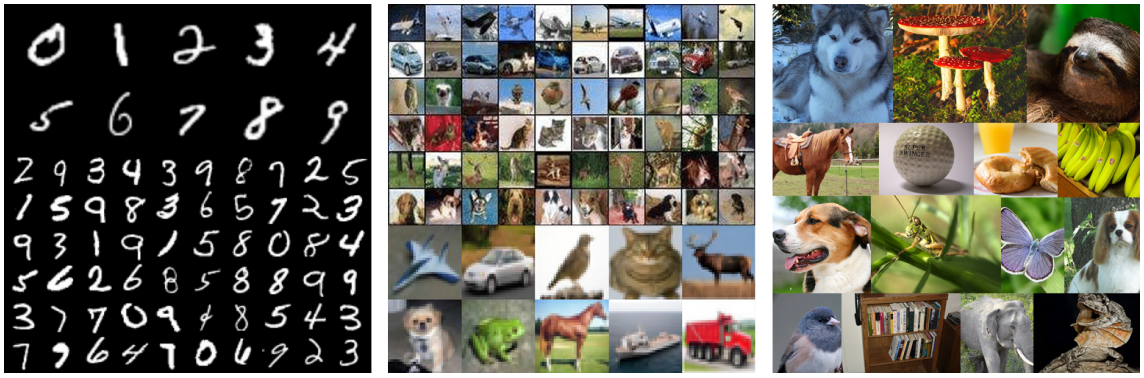


Figura 2.3: Imágenes de ejemplo de los tres datasets presentados: MNIST, CIFAR-10 e ImageNet respectivamente.

Capítulo 3

Ataques Adversarios

Como vimos en el capítulo anterior, las redes neuronales son modelos altamente expresivos capaces de obtener excelentes resultados en múltiples tareas de aprendizaje automático. Las redes convolucionales particularmente se destacan en la resolución de problemas de visión artificial como la clasificación de imágenes, detección de objetos y segmentación semántica.

Sin embargo se ha demostrado que son altamente vulnerables a determinados ejemplos de entrada, diseñados de forma específica para maximizar el error obtenido. Estos reciben el nombre de **ejemplos adversarios** y en general consisten en datos de entrada que han sido ligeramente modificados con el objetivo de engañar al modelo atacado [4]. Las **perturbaciones** realizadas por el atacante suelen ser sutiles, al punto de en muchos casos ser imperceptibles para un humano, pero aún así capaces de alterar de forma significativa la salida de la red. A los distintos algoritmos utilizados para generar este tipo de ejemplos se les conoce como **ataques adversarios**.

En el caso de redes de clasificación de imágenes, los ejemplos adversarios no son más que imágenes naturales a las que se le han modificado algunos, o incluso todos los píxeles con el objetivo de cambiar la clase predicha por la red. Estos se presentaron por primera vez en [4], atrayendo gran atención sobre el problema de los ejemplos adversarios (no solo para redes de clasificación sino para redes profundas en general), y este se ha mantenido como un tema activo de investigación desde entonces.

Formalmente, si tenemos una imagen de entrada válida $x \in \mathbb{R}^n$ y un clasificador tal que $C(x)$ es la clase asignada para x y $C^*(x)$ su clase correcta, decimos que x' es un ejemplo adversario si cumple que $C(x') \neq C^*(x)$ y x' se asemeja a x. Luego, de forma similar a como medimos el desempeño de la red según el porcentaje de imágenes clasificadas correctamente, se suele medir la efectividad del ataque con el porcentaje de ejemplos perturbados que logran engañar a la red, valor conocido como **fooling-rate** (tasa de engaño) del ataque.

Hoy en día existen numerosas soluciones basadas en aprendizaje profundo, muchas de ellas aplicadas a problemas de carácter crítico o sensible, motivo por el cual la seguridad se ha convertido en un factor sumamente importante para este tipo de sistemas. Sin embargo, debido a la naturaleza de las redes profundas, resulta muy difícil estudiar su **robustez** desde un punto de vista analítico, por lo que en la práctica se suelen usar ataques adversarios como una forma de evaluar qué tan resistentes son ante perturbaciones en los datos de entrada.

El estudio de los ejemplos adversarios también ayuda a comprender mejor el mismo fun-

cionamiento de las redes, aunque varias cuestiones relacionadas a estos aún permanecen sin resolver, como el desarrollo de redes inmunes a este fenómeno o siquiera determinar si esto es posible.

En las próximas secciones profundizamos en el estudio de dicho problema. Presentamos cómo se obtienen los ejemplos adversarios y qué características poseen, además de algunos de los ataques más relevantes publicados hasta el momento y las principales hipótesis que intentan explicar su existencia. Finalmente, terminamos el capítulo con un estudio de los mecanismos de defensa ante ejemplos adversarios, nuevamente presentando las características y algoritmos existentes.

Como ya mencionamos, nos centramos en ataques y defensas para redes de clasificación de imágenes. De todas formas, muchos de los conceptos presentados no son específicos a este tipo de redes y aplican tanto a los ataques como defensas para redes neuronales en general.

3.1. Generación de ejemplos adversarios

Si bien existen una gran variedad de algoritmos diseñados para generar ejemplos adversarios, la mayoría de ellos se basan en la misma idea. Aprovechan que las redes son diferenciables para calcular el gradiente de cierta función de error (como puede ser *cross-entropy loss*) con respecto a la entrada y luego la modifican en la dirección del gradiente, maximizando así el error de predicción de la red. Para calcular el gradiente se utiliza el mismo mecanismo empleado en el entrenamiento de los modelos, *backpropagation*[1], pero en lugar de propagar hasta los parámetros de la red se lo hace hasta los valores de la imagen de entrada.

Como vimos en el capítulo 2, durante la etapa de entrenamiento se ajustan los parámetros del modelo para minimizar la función de pérdida utilizada. Los ataques que estaremos estudiando, sin embargo, se dan en tiempo de evaluación, es decir, luego de finalizado el entrenamiento. En estos casos los parámetros del modelo han sido fijados y únicamente se pueden alterar los datos de entrada.

Para generar un ejemplo adversario el atacante define una función de pérdida análoga a la usada durante el entrenamiento (puede incluso usar la misma función) y calcula su gradiente con respecto a la entrada, en nuestro caso una imagen limpia. Este gradiente posee información de cómo impactan los cambios en la entrada sobre el resultado de la función definida, por lo que el atacante puede usarlo para modificar dicha entrada aumentando así el error producido por la red.

Volviendo al caso de clasificación de imágenes y teniendo en cuenta que la mayoría de los ataques buscan perturbar la entrada lo mínimo posible, se suelen generar los ejemplos adversarios resolviendo el siguiente problema de optimización:

$$\begin{aligned} \text{minimizar} \quad & D(x, x+r) \\ \text{sujeto a} \quad & C^*(x) \neq C(x+r) \\ & x + r \in [0,1]^n \end{aligned} \tag{3.1}$$

Donde D es cierta métrica de distancia que indica qué tan similar es la imagen perturbada con la original, y la restricción de $x + r \in [0, 1]^n$ refiere a que $x + r$ sea una imagen válida con valores de píxeles entre cero y uno [4].

[1] Algoritmo utilizado para computar el gradiente de la función de perdida con respecto a los parámetros de la red para un vector de entrada.

En este caso, únicamente se busca que la clase predicha para el ejemplo alterado sea distinta a la clase correcta, lo que corresponde a un ataque **no dirigido**. Como vemos más adelante, también existen los llamados **ataques dirigidos** en los cuales se busca una perturbación r tal que $C(x + r) = t$ para cierta clase objetivo $t \neq C^*(x)$.

Cabe aclarar que si bien la gran mayoría de ataques dependen del gradiente para generar ejemplos adversarios, se han propuesto algunas alternativas que no requieren este nivel de conocimiento sobre la red atacada (sin contar los ejemplos transferidos que veremos en próximas secciones), sino que logran engañarlas utilizando nada más que su salida, ya sea un vector de probabilidades [13] o únicamente la clase final [14].

3.2. Métricas de distancia

Con el objetivo de encontrar ejemplos adversarios tan parecidos como sea posible a las imágenes originales, muchos ataques suelen minimizar o incluir de alguna forma en el proceso de optimización un componente correspondiente a la distancia entre la imagen limpia y la perturbada. De esta forma logran maximizar el error de predicción de la red al mismo tiempo que minimizan el "tamaño" de la perturbación.

Idealmente nos gustaría que la métrica de distancia usada fuese un indicador de qué tan similares lucen dos imágenes a simple vista; sin embargo no existe una función que represente exactamente la percepción humana, y menos aún que lo haga de forma rápida y eficiente, por lo que en la práctica se suelen usar las denominadas normas L_p.

Las normas L_p, denotadas $\|\cdot\|_p$ se definen como:

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

Entre estas normas, las más usadas por los ataques son las L_0, L_2 y L_∞, con las cuales se obtienen ejemplos adversarios de características ligeramente diferentes. En la figura 3.1 podemos ver algunos ejemplos adversarios generados para cada una de estas normas.

- L_0: Esta norma corresponde a la cantidad de píxeles modificados por el ataque. Los ataques que minimizan esta norma suelen modificar pocos píxeles de la imagen original, pero con una diferencia de mayor magnitud.
- L_2: Esta penaliza los píxeles modificados en proporción a qué tanto difieren de su valor original. Con esta norma se tienden a realizar pequeñas modificaciones a muchos píxeles.
- L_∞: La norma infinito mide el máximo cambio realizado a cualquiera de los píxeles. Los ataques optimizados para esta norma suelen modificar todos los píxeles de la imagen en un valor menor o igual a cierta cota definida.

Otra métrica usada para medir el desempeño de los ataques (teniendo en cuenta que se suelen buscar perturbaciones imperceptibles) es la denominada *normalized dissimilarity*. Esta se define en función de las normas L_p y para un conjunto de ejemplos según la siguiente fórmula:

$$\frac{1}{N} \sum_{i=1}^{N} \frac{\|x_n - x'_n\|_p}{\|x_n\|_p} \tag{3.2}$$

Se busca que los ataques tengan una alta tasa de engaño o *fooling-rate*, al mismo tiempo que logren mantener una *normalized dissimilarity* baja.

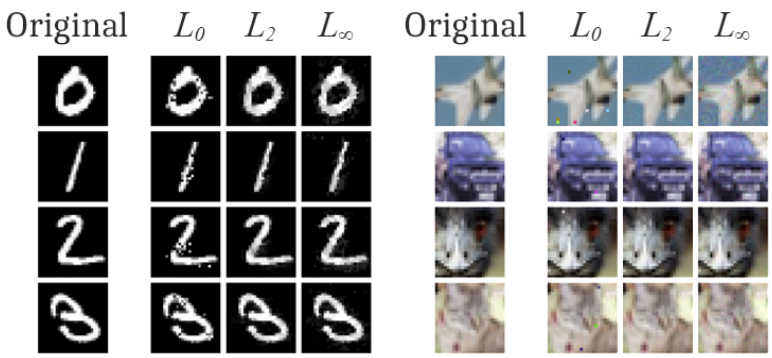


Figura 3.1: Ejemplos adversarios generados por el ataque Carlini&Wagner [15] sobre los *datasets* MNIST y CIFAR-10. En la primera columna de cada *dataset* se presentan imágenes limpias, las siguientes tres presentan ejemplos adversarios optimizados para las normas L_0, L_2 y L_∞ respectivamente.

3.3. Tipos de Ataques

En la actualidad existen una gran variedad de ataques capaces de generar ejemplos adversarios. En esta sección veremos una serie de criterios usados comúnmente para clasificarlos, al mismo tiempo que presentamos algunas de sus principales propiedades. Estos criterios nos serán de particular utilidad al momento de estudiar los ataques destacados presentados en la sección 3.4.

Etapa en la cual se realiza el ataque

Como mencionamos en el capítulo 2, los modelos deben ser entrenados con una gran cantidad de datos antes de poder ser usados para realizar predicciones. Por lo tanto, si consideramos que los ataques se dan a través de los ejemplos de entrada, un primer criterio según el cual los podemos clasificar es dependiendo de en qué etapa se realiza el ataque. Esto da lugar a los siguientes dos tipos:

- **Ataques de envenenamiento**: Estos se realizan durante la etapa de entrenamiento de la red. Consisten en introducir ejemplos alterados al *dataset* de entrenamiento con el objetivo de comprometer el proceso de aprendizaje.
- **Ataques de evasión**: Estos ataques se dan en tiempo de evaluación (luego de entrenada la red) y consisten en alterar los ejemplos de entrada para engañar de cierta forma al sistema. Son el tipo más común y en este informe nos estaremos refiriendo exclusivamente a este tipo de ataques.

Especificidad del ataque

Este segundo criterio tiene que ver con el objetivo del atacante, en si nada más busca generar un ejemplo que sea clasificado de forma incorrecta por la red o si además desea que este sea clasificado con una clase específica. Nuevamente consideramos dos tipos posibles:

- **Ataques no dirigidos**: Son los ataques más simples: el objetivo del atacante es hacer que el modelo realice una predicción equivocada, independientemente de cuál sea la clase predicha.

- **Ataques dirigidos**: Estos ataques son más complejos pero al mismo tiempo más potentes que los anteriores. En este caso el atacante no solo busca que la salida del modelo sea una clase equivocada, sino que además sea una clase específica previamente definida. Claramente este tipo de ataques proponen un mayor riesgo dado que permiten controlar la salida de la red, aunque suelen requerir perturbaciones de mayor magnitud.

También relacionado a la especificidad del ataque pero desde otro enfoque, podemos clasificar a los algoritmos según si estos generan perturbaciones específicas para una imagen dada o si las generan de forma que funcionen sobre una variedad de imágenes. De esta forma identificamos los siguientes:

- **Ataques específicos**: Estos abarcan a la mayoría de ataques existentes. El algoritmo recibe una imagen limpia y la modifica generando un ejemplo adversario que se asemeja a la imagen original. En general nos referiremos a este tipo de ataques siempre que no se aclare explícitamente lo contrario.
- **Ataques universales**: Los ataques universales, a diferencia de los específicos, no generan ejemplos adversarios de forma directa sino que se encargan de generar perturbaciones agnósticas a las imágenes sobre las cuales serán aplicadas. Fueron introducidos en [9] donde los autores muestran la existencia de las denominadas **perturbaciones universales**. Estas son básicamente imágenes que a simple vista lucen como ruido aleatorio, pero que al ser superpuestas sobre imágenes naturales del *dataset* causan que estas sean clasificadas de forma incorrecta con alta probabilidad.

Medio del ataque

Los primeros ataques desarrollados se basaban en modificar de forma directa la entrada proporcionada a la red para atacarla. Sin embargo, estudios posteriores demostraron que también es posible trasladar ejemplos adversarios al mundo físico, generando imágenes que logran permanecer adversarias aún luego de ser impresas y percibidas por sensores [6]. Esto da lugar a un nuevo grupo de ataques con distintas características, los cuales proponen un riesgo aún un mayor para los sistemas basados en aprendizaje automático existentes en la actualidad.

- **Ataques en el mundo digital**: Como mencionamos, en este tipo de ataques se asume que el atacante es capaz de proporcionarle datos de forma directa a la red. El conocer de forma exacta cómo serán percibidas las imágenes por la red le permite al atacante generar perturbaciones óptimas, que maximicen el error de predicción aún con normas muy pequeñas. En este trabajo asumimos este tipo de ataques si no se especifica de otra forma.
- **Ataques en el mundo físico**: Estos ataques por otro lado se enfocan en generar perturbaciones que se transfieran con mayor probabilidad al mundo físico. Para esto deben tener en cuenta los distintos factores que puedan afectar su funcionamiento, que en el caso de sistemas de visión artificial incluyen desde el ruido introducido al imprimir las perturbaciones hasta el introducido por las cámaras, además de cambios en la iluminación, ángulo de visión, etc.

Conocimiento del atacante

Uno de los principales factores al momento de definir un ataque es la cantidad de información que se tiene sobre el sistema atacado. Esta, sumada a otros aspectos como el control

que posee el atacante componen lo que se conoce normalmente como **modelo de amenaza** del ataque. Dentro de los datos relevantes que el atacante puede conocer se encuentran la arquitectura de la red, algoritmo de entrenamiento y *dataset* utilizado, etc. Para referirse a los distintos niveles de conocimiento que asumen los algoritmos se suelen utilizar las siguientes dos categorías:

- **white-box**: En esta modalidad se asume que el atacante tiene un conocimiento completo sobre el sistema a atacar. Esto incluye los datos ya mencionados, además de otros como los parámetros del modelo entrenado, mecanismo de defensa en caso de haberlo, aleatoriedad en tiempo de evaluación (sean los valores aleatorios o su distribución) y cualquier otra información relevante. Representa el caso más favorable para el atacante y por lo tanto el más difícil de defender. En general los ataques que hacen uso del gradiente funcionan en esta modalidad, a excepción como veremos a continuación de los ataques por transferencia.
- **black-box**: Los ataques *black-box*, en contraste, no asumen conocimiento ninguno sobre el modelo atacado. En este caso los atacantes están limitados a usar el modelo víctima como un **oráculo** al cual pueden proporcionar ejemplos de entrada para luego observar su salida, la cual a su vez puede tratarse de la clase asignada por la red, la confianza sobre dicha clase o el vector de probabilidades completo. Como ya mencionamos, existen algoritmos que utilizan únicamente la salida proporcionada por la red atacada; sin embargo la gran mayoría de ataques de esta modalidad se basan en otra de las características importantes de los ejemplos adversarios, la de **transferibilidad** (descrita más adelante).

Si bien estos términos ayudan a definir de forma rápida y sencilla las capacidades del atacante, en algunos casos pueden resultar poco exactos dando lugar a ambigüedades, por lo que en general es recomendable realizar una descripción detallada del modelo de amenaza considerado. Esto también aplica para las defensas dado que es de suma importancia definir de forma clara las condiciones bajo las que estas argumentan seguridad.

Transferibilidad

La transferibilidad refiere a la capacidad que poseen los ejemplos adversarios para engañar múltiples redes diferentes. Se ha demostrado que, tras generar ejemplos adversarios sobre una red determinada, por ejemplo mediante técnicas de tipo *white-box*, un gran porcentaje se mantienen adversarios al ser trasladados a otras redes, incluso de arquitecturas diferentes y entrenadas sobre conjuntos de datos disjuntos [4, 16].

Esta propiedad es fundamental debido a que le permite al atacante usar **modelos sustitutos** para aplicar técnicas *white-box* más potentes, y luego transferir los ejemplos generados a sistemas de los que se posee poco o ningún conocimiento. En la tabla 3.1 se presenta como ejemplo la efectividad de los ejemplos generados por el ataque MI-FGSM [17] tras ser transferidos a varias arquitecturas de redes neuronales.

La transferibilidad claramente supone un problema importante para las soluciones existentes basadas en aprendizaje automático, ya que mantener su información relevante de forma oculta no basta para mantenerlas a salvo de este tipo de ataques.

Costo computacional

Si consideramos el costo computacional de generar ejemplos adversarios, uno de los factores que más influye es el número de iteraciones que realiza el algoritmo. Esto refiere a la

	Inc-v3	Inc-v4	IncRes-v2	Res-152
Inc-v3	**100.0**	48.8	48.0	35.6
Inc-v4	65.6	**99.9**	54.9	46.3
IncRes-v2	69.8	62.1	**99.5**	50.6
Res-152	53.6	48.9	44.7	**98.5**

Cuadro 3.1: *Fooling rates* obtenidos por el ataque MI-FGSM (presentado en la sección 3.4.2) sobre los modelos Inc-v3, Inc-v4, IncRes-v2 y Res-152 con sus respectivos porcentajes de transferibilidad.

cantidad de veces que se procesan las entradas o, en la mayoría de los casos, la cantidad de veces que se calcula el gradiente. Es posible separar los algoritmos en:

- **Ataques de un paso**: Estos realizan un único cálculo del gradiente por cada ejemplo adversario generado. Tienen la ventaja de que se pueden ejecutar de forma muy eficiente, lo que los hace atractivos para su uso en **entrenamiento adversario**, un mecanismo de defensa que veremos en próximas secciones. Además, se ha visto que los ejemplos generados con este tipo de algoritmos suelen transferirse a modelos distintos con mayor probabilidad de éxito que los ejemplos generados mediante ataques iterativos [18].

- **Ataques iterativos**: Requieren múltiples evaluaciones del gradiente (o de la imagen en ciertos casos) para generar las perturbaciones. Son considerablemente más lentos que los ataques de un paso pero capaces de alcanzar mayores *fooling-rates* con menores perturbaciones en la modalidad *white-box*. Estos algoritmos suelen resolver problemas de optimización por lo que es de esperar que los métodos iterativos encuentren mejores soluciones.

3.4. Ataques destacados

En esta sección presentamos algunos de los ataques más relevantes publicados hasta el momento. Mediante su estudio podemos conocer las principales estrategias utilizadas por los autores para encontrar ejemplos adversarios, al mismo tiempo de ver algunos casos para las distintas categorías presentadas en la sección anterior.

Nuestro objetivo no es el de dar una descripción en extremo detallada de cada uno de los métodos seleccionados, sino presentar en términos generales sus funcionamientos, cómo difieren de otros ataques previos y cuáles son sus principales aportes. Para una visión más completa alentamos al lector a ver las publicaciones referidas en cada caso.

Para definir estos métodos usaremos la siguiente notación:

x	Imagen de entrada natural, vectores de $\mathbb{R}^n$
r	Perturbación adversaria aplicada a las imagen x
x^{adv}	Ejemplo adversario, resultado de agregar la perturbación a la imagen: $x^{adv} = x + r$

$F(x)$	Vector de probabilidades resultante de la función de clasificación de la red para la imagen x
$C(x)$	Clase asignada a x por la red, correspondiente al elemento de mayor probabilidad en $F(x)$
y	Clase correcta de la imagen x
t	Clase objetivo para el caso de ataques dirigidos
$J(x, y)$	Función de error de la red para la imagen x y clase y

3.4.1. L-BFGS

Al presentar los ejemplos adversarios por primera vez en [4], los autores propusieron además un método para generarlos. En este plantean una variación al problema de optimización presentado en la ecuación 3.1 para el caso dirigido, al cual proponen resolver aplicando una versión restringida del método L-BFGS [19].

El algoritmo consiste en aplicar *line-search* para encontrar el menor $c > 0$ tal que la perturbación r obtenida al minimizar la siguiente expresión cumpla que $C(x + r) = t$ para la clase objetivo t.

$$\begin{array}{ll} \underset{r}{\text{mínimo}} & c\,\|r\|_2 + J(x + r, t) \\ \text{sujeto a} & x + r \in [0, 1]^n \end{array}$$

De esta forma se obtiene un método iterativo y dirigido en modalidad *white-box* capaz de generar perturbaciones casi imperceptibles que logran engañar a los modelos evaluados en el 100 % de los casos, según reportan sus autores.

3.4.2. FGSM y variantes

El *Fast Gradient Sign Method* (FGSM)[5] es uno de los primeros ataques publicados y a su vez uno de los más simples. Se trata de un ataque de un paso y no dirigido (aunque es posible adaptarlo a dirigido) optimizado para la norma L_∞.

Este método propone utilizar el gradiente de la función de pérdida J con respecto a la entrada x, o, más precisamente, el signo de sus elementos (1 si son positivos o -1 si son negativos) para generar una perturbación de la siguiente manera:

$$r = \epsilon\,\text{sign}(\nabla_x J(x, y))$$

De esta forma se obtienen perturbaciones con una norma L_∞ acotada, regulada por el valor del hiperparámetro ϵ. Los autores reportan que usando un ϵ de 0,25 fueron capaces de obtener *fooling rates* de casi 100 % en redes entrenadas sobre el *dataset* MNIST[2].

Este algoritmo muestra que es posible generar ejemplos adversarios efectivos y poco perceptibles de forma muy eficiente. A continuación presentamos algunos algoritmos inspirados en el FGSM que buscan mejorar su desempeño en distintos aspectos: *fooling-rate*, magnitud de las perturbaciones y transferibilidad.

[2]Para valores de píxeles en el rango $[0, 1]$.

R+FGSM

El *Random FGSM* (R+FGSM) [20] propone una estrategia muy sencilla para generar perturbaciones superiores a las obtenidas por el algoritmo original. Consiste en aplicar FGSM pero partiendo de un punto aleatorio cercano a x en vez de directamente sobre él.

La idea detrás de esto surge de las evaluaciones realizadas por los autores en las cuales observan que la función de error $J(x, y)$ presenta curvaturas bruscas alrededor de los ejemplos del *dataset*, las cuales pueden enmascarar la dirección real en la cual se maximiza dicha función. Para lidiar con esto proponen realizar un desplazamiento aleatorio previo al cálculo del gradiente con la intención de escapar de estas regiones irregulares y obtener un vector que aproxime mejor la dirección de mayor pérdida.

Luego proponen un nuevo método de un paso, capaz de aplicarse eficientemente y que según lo reportado presenta mejores resultados que el método original. Este hace uso de un hiperparámetro adicional $\alpha < \epsilon$ y se formula de la siguiente manera:

$$\begin{array}{rrcl} & x^{adv} & = & x' + (\epsilon - \alpha) \cdot sign(\nabla_{x'} J(x', y)) \\ \text{donde} & x' & = & x + \alpha \cdot sign(\mathcal{N}(0^n, 1^n)) \end{array}$$

Las perturbaciones generadas, al igual que en FGSM, poseen una norma L_∞ acotada por ϵ con la diferencia de que parte de la perturbación proviene del desplazamiento aleatorio y otra parte de la dirección del gradiente.

BIM

El *Basic Iterative Method* (BIM) [6] es otro método basado en el FGSM, pero en este caso se trata de un método iterativo. Consiste en una extensión bastante directa del algoritmo original en la cual básicamente se aplica FGSM múltiples veces escalando el vector del signo por cierto parámetro α en cada iteración y haciendo *clipping*[3] del ejemplo resultante para mantenerlo dentro de la esfera de radio ϵ y centro x.

$$\begin{array}{rcl} x_0^{adv} = & x \\ x_{i+1}^{adv} = & Clip_{x,\epsilon}(x_i^{adv} + \alpha \cdot sign(\nabla_x J(x_i^{adv}, y)) \end{array}$$

En los experimentos reportados, este método logra obtener mejores *fooling rates* que el FGSM original para valores bajos de ϵ, demostrando ser mejor al momento de encontrar perturbaciones pequeñas y efectivas. Sin embargo esta ventaja disminuye a medida que se les permite a ambos ataques realizar mayores perturbaciones, siendo el BIM menos efectivo para valores altos de ϵ.

MI-FGSM

El *Momentum Iterative FGSM* (MI-FGSM) [17] es aún otro método iterativo no dirigido que a su vez extiende al BIM presentado previamente. Los autores proponen aplicar **momentum** (inercia) al momento de maximizar la función de pérdida, una técnica utilizada para acelerar y estabilizar algoritmos de descenso por gradiente mediante el uso de un vector de velocidad que se construye acumulativamente en la dirección del gradiente.

[3] Clipping en este contexto consiste en la acción de acotar una o muchas variables manteniéndolas dentro de cierto rango. Dado un centro c, un radio positivo r y una variable x, se define $Clip_{c,r}(x)$ como $c + r$ si $x > c + r$, $c - r$ si $x < c - r$ o x de lo contrario.

Sea ϵ la norma máxima de la perturbación, N el número de máximo de iteraciones, g el vector de velocidad y μ el factor de envejecimiento, se definen los ejemplos de la siguiente forma:

$$g_0 = 0,\ x_0^{adv} = x,\ \alpha = \frac{\epsilon}{T} \tag{3.3}$$

$$g_{n+1} = \mu \cdot g_n + \frac{\nabla_x J(x_t^{adv}, y)}{\left\|\nabla_x J(x_t^{adv}, y)\right\|_1} \tag{3.4}$$

$$x_{n+1}^{adv} = x_t^{adv} + \alpha \cdot sign(g_{n+1}) \tag{3.5}$$

Como se ve en las ecuaciones 3.4 y 3.5, en cada iteración del algoritmo se actualiza el ejemplo en la dirección del vector g_n, el cual acumula información de los últimos n gradientes en contraste al BIM, que se basa únicamente en el último.

Los autores argumentan que el FGSM genera perturbaciones sub-óptimas que no se ajustan lo suficiente al modelo atacado y por lo tanto no logran alcanzar un buen muy desempeño. El BIM por otro se comporta de forma demasiado *greedy*, generando ejemplos que convergen fácilmente a máximos locales específicos al modelo y no logran transferirse a otros modelos.

El MI-FGSM en cambio propone ser un método capaz de generar perturbaciones poderosas y con alta transferibilidad, siendo efectivo tanto para atacar a modelos conocidos en una modalidad *white-box* como para atacar a modelos diferentes en un estilo *black-box*.

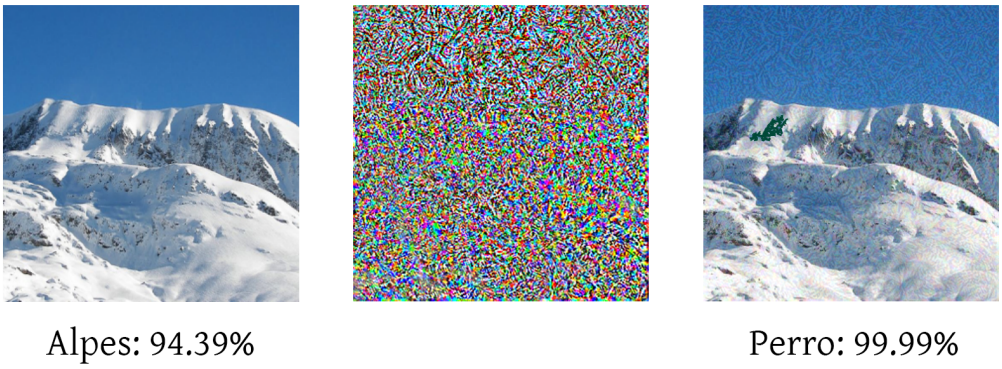


Figura 3.2: Ejemplo adversario generado con el método MI-FGSM. En esta se puede observar la imagen limpia seguida de la perturbación adversaria y el ejemplo resultante, indicando en ambos casos la clase asignada por la red.

3.4.3. JSMA

El *Jacobian-based Saliency Map Attack* (JSMA) [21] es un ataque iterativo y dirigido optimizado para norma L_0, por lo que busca generar ejemplos adversarios modificando la menor cantidad de píxeles posible. En términos generales, este algoritmo procede iterativamente determinando en cada iteración el par de píxeles que influyen en mayor medida sobre la clasificación final de la red.

Formalmente, sea $Z(x)$ el vector con los valores asignados a cada clase (previo a la conversión en probabilidades) y $Z_j(x)$ el valor asignado a la clase j; se busca modificar x de

tal manera que el valor de $Z_t(x)$ aumente para la clase objetivo t y los valores de $Z_j(x)$ disminuyan para las otras clases $j \neq t$.

Para identificar el par de píxeles mencionados se comienza por calcular el jacobiano de $Z(x)$ con respecto a la entrada. Posteriormente se genera el denominado *adversarial saliency map* en función de pares de píxeles p y q, con el cual finalmente se determinan los píxeles a modificar siguiendo la siguiente ecuación:

$$(p^*, q^*) = \arg\max_{(p,q)}(-\alpha_{pq} \cdot \beta_{pq}) \cdot (\alpha_{pq} > 0) \cdot (\beta_{pq} < 0)$$

Donde:

$$\begin{aligned} \alpha_{pq} = & \sum_{i \in \{p,q\}} \frac{\partial F_t(x)}{\partial x_i} \\ \beta_{pq} = & \sum_{i \in \{p,q\}} \sum_{j \neq t} \frac{\partial F_j(x)}{\partial x_i} \end{aligned}$$

De esta manera se selecciona el par de píxeles que al ser incrementados influyen positivamente sobre el valor de la clase objetivo t (componente α) y negativamente sobre los valores de las otras clases (componente β), y que además lo hacen en mayor proporción.

El algoritmo completo, por lo tanto, consiste en generar el *adversarial saliency map* en cada iteración para seleccionar el par de píxeles de mayor relevancia. Luego se incrementa el valor de dichos píxeles según un hiperparámetro θ y finalmente se los remueve del conjunto de píxeles considerados para las próximas iteraciones (este conjunto inicialmente se compone por todos los píxeles de la imagen).

El algoritmo finaliza cuando se cumple una de las siguientes dos condiciones:

- La imagen perturbada es clasificada como la clase objetivo: $C(x^{adv}) = t$.
- Se alcanza el máximo número de píxeles modificados, regulado por otro hiper-parámetro γ. En caso de no haber un límite, se detiene en caso de haber modificado todos los píxeles.

Los autores evalúan este ataque sobre el *dataset* MNIST, en el cual reportan lograr engañar a la red en un 97 % de los casos modificando únicamente un 4 % de los píxeles. Si bien logra obtener buenos resultados este método tiene la desventaja de ser considerablemente demandante, con un costo que crece exponencialmente con la dimensión de la entrada, haciéndolo inviable para imágenes de mayor tamaño como las de ImageNet.

3.4.4. DeepFool

DeepFool [22] es un método iterativo y no dirigido que toma un enfoque geométrico para buscar las perturbaciones óptimas en cada caso, utilizando para esto la norma L_2. Dado que su formulación exacta es particularmente compleja, nos centraremos en presentar la idea general detrás del algoritmo.

Sea y la clase correcta de x; este método comienza por considerar la región P de puntos alrededor de x que son clasificados con su misma clase:

$$P = \{x^* : C(x^*) = y\}$$

Definida esta región, la perturbación buscada r se corresponde al vector de menor norma que traslada a x hasta el complemento de P. Para encontrar este vector los autores

comienzan por considerar el caso lineal en el cual P es un poliedro y r es el vector que proyecta a x sobre el hiperplano más cercano de los límites de P.

Luego extienden la solución diseñada a clasificadores no lineales. En estos casos la región P ya no es un poliedro, por lo que el algoritmo propone proceder de forma iterativa aproximando P en cada iteración por el poliedro $\tilde{P}_i$ como se muestra en la figura 3.3. De esta forma, el ejemplo adversario se calcula proyectando sucesivamente la imagen inicial x hacia los complementos $\tilde{P}_i^c$ hasta que finalmente se logra cambiar la clasificación de la red.

De esta forma se logran generar ejemplos adversarios realizando perturbaciones prácticamente imperceptibles, muy superiores a las generadas por otros métodos existentes hasta el momento.

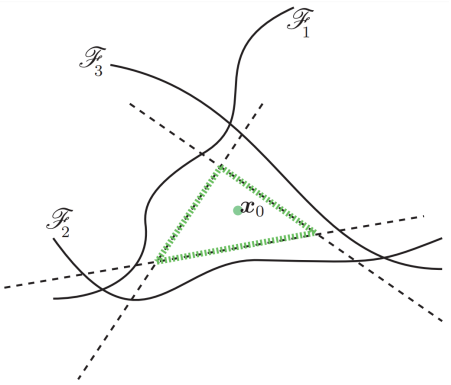


Figura 3.3: Representación de la región P considerada por el método *DeepFool* y su aproximación mediante el poliedro $\tilde{P}$ alrededor de un punto x_0.

3.4.5. Universal Adversarial Perturbations

Se conoce como *Universal Adversarial Perturbations* al algoritmo presentado en [9] que, a diferencia de los otros métodos presentados hasta el momento, busca generar perturbaciones universales. Es un método iterativo y no dirigido que, como veremos a continuación, no es específico a una norma en particular.

Formalmente, el objetivo de este algoritmo es el de encontrar perturbaciones r^* de norma acotada por cierto ϵ que logren cambiar la clasificación de la red al ser aplicadas sobre distintas imágenes, con una probabilidad regulada por cierto δ:

- $\|r^*\|_p \leq \epsilon$
- $\underset{x\sim\mu}{\mathbb{P}}\big(C(x+r^*) \neq C(x)\big) \geq 1-\delta$

Donde μ corresponde a la distribución de imágenes naturales.

Esta perturbación universal r^* es construida de forma incremental a partir de un conjunto de imágenes x_i. El algoritmo itera sobre las imágenes calculando en cada caso la perturbación mínima Δr_i que sumada a la perturbación actual logre cambiar la clasificación de la red como se ilustra en la figura 3.4.5.

$$\Delta r_i = \arg\min_r \|r\|_p \quad \text{sujeto a} \quad C(x_i + r^* + r) \neq C(x_i) \tag{3.6}$$

Luego de cada iteración se actualiza r^* agregándole el valor de Δr_i y proyectando el vector resultante para mantener su norma por debajo de la cota especificada. Este procedimiento se repite con todas las imágenes x_i, potencialmente más de una vez con cada una, hasta alcanzar el *fooling rate* deseado.

Una particularidad de este ataque es que es agnóstico al algoritmo utilizado en la ecuación 3.6 para encontrar la perturbación mínima, siendo *DeepFool* el optado por los autores en el trabajo original. De esta forma logran generar perturbaciones sutiles pero capaces de engañar a las redes evaluadas en un gran porcentaje de las imágenes del *dataset* de prueba, entre un 80 % y 90 % de ellas aproximadamente.

Dichas perturbaciones no sólo alcanzan buenos *fooling rates* sobre el modelo en el cual fueron generadas, sino que también se comprobó que se trasladan con alta probabilidad a otros modelos, con *fooling rates* que varían entre el 40 % y 70 %.

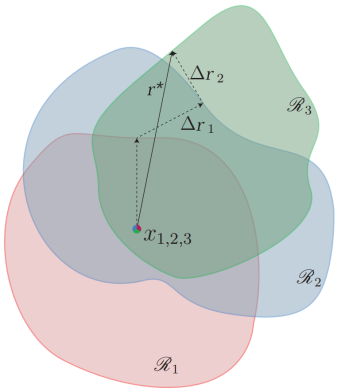


Figura 3.4: Representación del algoritmo *Universal Adversarial Perturbations*. En esta se representa el vector de perturbación r^* generado a partir de tres imágenes x_1, x_2 y x_3 las cuales se muestran superpuestas por simplicidad.

3.4.6. Carlini&Wagner

Carlini&Wagner es el nombre usado para referirse a los tres métodos propuestos en [15], que consisten en potentes algoritmos iterativos y dirigidos optimizados para las distintas normas L_0, L_2 y L_∞. Estos parten del problema de optimización presentado en las ecuaciones 3.1 para el caso dirigido y proponen reformularlo de alguna manera que sea más apropiada para resolverse mediante algoritmos de optimización existentes.

En primer lugar sustituyen la restricción de $C(x+r) = t$ por una función objetivo f que cumpla la siguiente condición:

$$C(x+r) = t \iff f(x+r) \leq 0$$

Se evalúan varias opciones para f optando finalmente por la que obtiene mejores resultados en la práctica:

$$f = (\max_{i \neq t}(Z(x')_i) - Z(x')_t)^+$$

donde $(v)^+ = max(v, 0)$.

Una vez seleccionada f es posible replantear el problema original de la siguiente manera:

$$\begin{array}{rl} \text{minimizar} & \|r\|_p + c \cdot f(x+r) \\ \text{sujeto a} & x + r \in [0,1]^n \end{array} \tag{3.7}$$

Además de introducir las normas L_p como métricas de distancia se agrega una constante c que ayuda a regular el algoritmo para que se minimice la norma de la perturbación y la función objetivo simultáneamente, en lugar priorizar un sumando sobre el otro.

Luego, para contemplar la restricción de $x + r \in [0,1]^n$ se propone realizar el siguiente cambio de variable, que introduce la variable w para obtener resultados que sean automáticamente válidos.

$$r = \frac{1}{2}(\tanh(w) + 1) - x$$

De esta forma, el problema de encontrar la perturbación mínima se reduce a minimizar la siguiente función:

$$||\frac{1}{2}(\tanh(w) + 1) - x||_p + c \cdot f(\frac{1}{2}(\tanh(w) + 1))$$

Para el caso de la norma L_2, esta ecuación puede ser minimizada mediante métodos tradicionales de optimización. Los autores utilizaron *Adam* [23], que según lo reportado logra converger más rápido que los otros algoritmos evaluados: descenso por gradiente tradicional y descenso por gradiente con *momentum*. Para las normas L_0 y L_∞ sin embargo se realizan los cambios descritos a continuación.

Variación para norma L_0

Dado que la norma L_0 no es diferenciable, se modifica el algoritmo propuesto para proceder de forma iterativa. Se parte de un conjunto que incluye todos los píxeles de la imagen y en cada iteración se genera un ejemplo adversario utilizando el algoritmo de L_2 sobre dichos píxeles.

Los ejemplos generados son utilizados para determinar el conjunto de píxeles con menor influencia sobre la clasificación de la imagen. Estos luego son removidos del conjunto de píxeles a considerar por el algoritmo, repitiendo este proceso hasta llegar al punto en el cual no es posible generar un ejemplo adversario con los píxeles disponibles. De esta forma se utiliza el algoritmo planteado sobre la menor cantidad de píxeles posible, minimizando así la norma L_0 de la perturbación.

Variación para norma L_∞

En el caso de L_∞ se observó que el descenso por gradiente no producía buenos resultados. Por este motivo los autores remplazaron el término correspondiente a la norma de la perturbación en la función objetivo por un término que penaliza valores por encima de cierto τ.

Luego, al igual que para la norma L_0 se procede iterativamente minimizando en cada iteración la siguiente ecuación:

$$c \cdot f(x+r) + \sum_i [(r_i - \tau)^+]$$

Si al finalizar una iteración todos los r_i son menores a τ entonces se reduce su valor por un factor de 0,9, de lo contrario se finaliza el algoritmo. De esta forma logran reducir progresivamente el valor de τ e indirectamente la norma L_∞ de la perturbación en cada iteración.

3.4.7. EOT

Los algoritmos descritos hasta el momento están diseñados para funcionar en el mundo digital. El *Expectation Over Transformation* (EOT) [24], por otro lado, es un ataque pensado para generar ejemplos adversarios que sean robustos ante las distintas variaciones introducidas al pasar al mundo físico.

Como mencionamos en la sección 3.3, los ejemplos en el mundo físico se ven sujetos a una serie de alteraciones, desde el ruido introducido por los sensores hasta la diferencia de puntos de vista, iluminación, etc. Este método propone modelar tales distorsiones dentro del proceso de optimización, definiendo una distribución T de transformaciones t.

Estas transformaciones t toman ejemplos x y les realizan ciertas modificaciones con el objetivo de simular la forma en la cual serían percibidos en el mundo real. Luego, en lugar de generar perturbaciones que maximicen el error para un único ejemplo, se buscan de tal manera que maximicen la esperanza del error de $t(x)$ para transformaciones de T, con el objetivo de que las perturbaciones resultantes sean resistentes a múltiples alteraciones. Entre las transformaciones modeladas en T se incluyen rotación, traslación, reescalado, ruido gaussiano, aclarado y oscurecimiento de imágenes.

Otra particularidad de este algoritmo es que busca que los ejemplos generados se asemejen a los originales en la forma que son percibidos por la red, por lo que considera la métrica de distancia sobre los ejemplos con la transformación incluida: $D(t(x), t(x^{adv}))$.

Finalmente, sea $F_y(x)$ la probabilidad asignada por el modelo a la clase objetivo y, se generan los ejemplos adversarios resolviendo el siguiente problema de optimización:

$$x^{adv} = \arg\max_{x'} \mathbb{E}_{t\sim T}[\log F_y(t(x'))]$$
$$\text{sujeto a } \mathbb{E}_{t\sim T}[D(t(x), t(x'))] < \epsilon$$

Para resolver esta optimización los autores plantean el problema de forma similar a la usada en el ataque *Carlini&Wagner* [15]. Luego evalúan los ejemplos generados contra 1000 transformaciones aleatorias de la distribución seleccionada, ante lo que reportan que los ejemplos permanecen adversarios en un 96 % de los casos.

No solo logran generar imágenes que permanecen adversarias luego de pasar al mundo físico sino que van más allá extendiendo el método para generar objetos tridimensionales que logran engañar a las redes de clasificación tras ser fotografiados como en el caso de la figura 3.5.

3.4.8. BPDA

El *Backward Pass Differentiable Approximation* (BPDA) [26] es un algoritmo orientado a encontrar ejemplos adversarios en modalidad *white-box* sobre redes en las cuales el gradiente no existe o es incorrecto, causado intencional o involuntariamente por ejemplo tras el uso de operaciones no diferenciables ó generando inestabilidad numérica. Este ataque puede

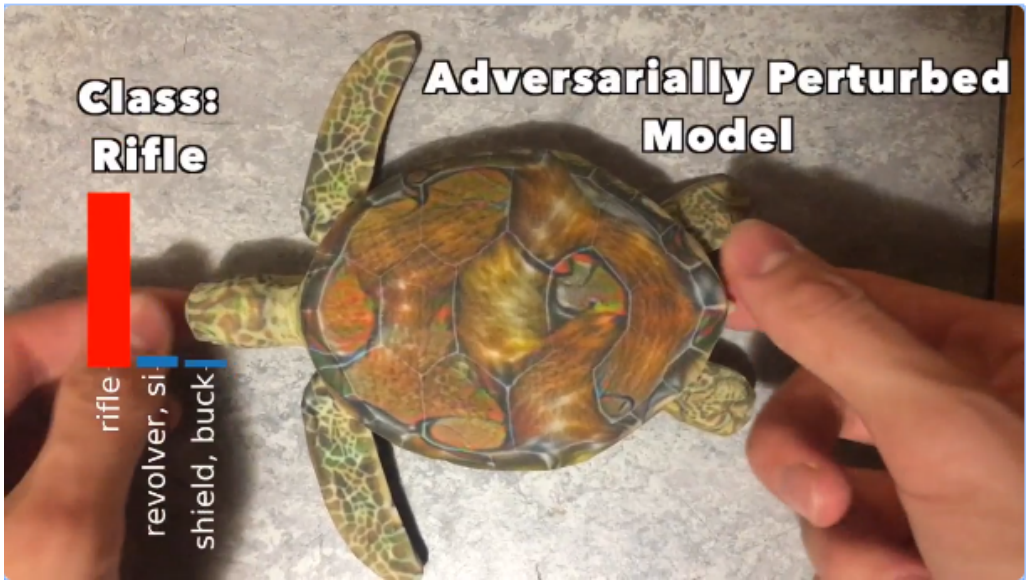


Figura 3.5: Captura de vídeo mostrando como la impresión 3D de una tortuga con una perturbación adversaria logra ser clasificada como rifle con alta certeza [25]

verse análogamente como una técnica para aproximar gradientes en los casos mencionados, permitiendo el uso de otros ataques tradicionales.

Los autores de este método notan que muchas de las defensas propuestas recientemente consisten en inutilizar el gradiente de la red con el objetivo de impedir la aplicación de técnicas *white-box*, lo que denominan **ofuscación de gradientes**, y argumentan que esto brinda una "falsa sensación de seguridad".

Comienzan por identificar tres escenarios en los cuales algunas defensas existentes hacen que los gradientes no aporten información de utilidad y luego proponen formas de sobrepasar cada una de ellas.

- **Shattered Gradients**: Este refiere a defensas no diferenciables o que de alguna forma causan que el gradiente no exista o sea incorrecto.
- **Stochastic Gradients**: Correspondiente a defensas que agregan aleatoriedad, ya sea en la misma red o en transformaciones aplicadas a la entrada.
- **Exploding & Vanishing Gradients**: Fenómeno causado por defensas que consisten en múltiples evaluaciones de una red, lo que puede verse de forma análoga como una red extremadamente profunda, en la cual los gradientes se desvanecen o se disparan.

Para el caso de defensas que aplican *stochastic gradients* proponen atacarlos usando EOT, a las de *exploding & vanishing gradients* argumentan que pueden ser atacadas reparametrizando funciones, y finalmente para el caso de *shattered gradients* proponen un método nuevo descrito a continuación.

Caso especial

El ataque propuesto ataca puntualmente el caso de *Shattered Gradients* para métodos que intencional o involuntariamente inutilizan el gradiente. Dentro de este se identifica un caso

especial correspondiente a las defensas que aplican cierta función no diferenciable g sobre la entrada para luego pasar su resultado al clasificador.

Con estas funciones se busca eliminar posibles perturbaciones adversarias presentes en la entrada y suelen definirse de forma tal que $g(x) \approx x$ para no deteriorar el desempeño original de los modelos. Luego, aprovechando que $g(x)$ es similar a x, el método propuesto consiste en aproximar el gradiente de g como el de la función identidad y consecuentemente aproximar el gradiente de $F(g(x))$ en el punto $\hat{x}$ de la siguiente manera:

$$\nabla_x F(g(x))|_{x=\hat{x}} \;\approx\; \nabla_x F(x)|_{x=g(\hat{x})}$$

Con esta aproximación del gradiente es posible aplicar cualquiera de los métodos presentados previamente, aún en redes que utilicen un pre-procesamiento no diferenciable.

Caso general

Finalmente extienden el algoritmo al caso general, en el cual el componente no diferenciable se puede encontrar en cualquier parte de la red. Para esto se considera al clasificador $F(\cdot)$ como la composición de capas $f^j(\cdot)$, donde f^i es una capa no diferenciable. Este método propone buscar una función h diferenciable que cumpla $h(x) \approx f^i(x)$ y luego para aproximar el gradiente $\nabla_x F(x)$ se debe realizar la pasada hacia delante en F utilizando f^i pero en la pasada hacia atrás se sustituye $f^i(x)$ por $h(x)$.

De esta forma el BPDA logra obtener aproximaciones del gradiente lo suficientemente buenas como para generar ejemplos adversarios, demostrando que el uso de funciones no diferenciables no supone un mecanismo de defensa robusto contra ataques *white-box*.

3.4.9. Otros ataques

Para culminar el estudio de ataques existentes presentaremos de forma muy breve algunos algoritmos adicionales. Con esto pretendemos nada más que introducir al lector a varios métodos que de cierta manera se diferencian de los otros ataques presentados y son una muestra de la variedad de enfoques presentados en un área donde la literatura crece rápidamente.

ZOO

El *Zeroth order optimization* [13] es otro ataque iterativo de modalidad *black-box*, pero que a diferencia de la mayoría de ataques en esta modalidad no requiere de un modelo sustituto. Este toma ideas del ataque Carlini&Wagner [15] en cuanto al planteamiento del problema de optimización pero no usa un modelo para calcular el gradiente, sino que lo aproxima utilizando únicamente la salida del modelo atacado.

Adversarial patch

El *Adversarial Patch* [27] es otro ataque universal diseñado para funcionar también en el mundo físico. Consiste en generar un "parche adversario" el cual al ser aplicado sobre un

objeto cualquiera logra alterar su clase ante el clasificador. A diferencia de ataques que modifican levemente todos los píxeles de la entrada, estos parches remplazan completamente parte de ella.

Para generar dichos parches se utiliza una variación del ataque EOT [24], en la cual además de considerar la distribución de transformaciones se tiene en cuenta un conjunto de imágenes y una distribución de ubicaciones para el parche.

One Pixel Attack

El *One Pixel Attack* [28] es un ataque optimizado para la norma L_0 que como dice su nombre, consiste en modificar únicamente un pixel de la imagen de entrada. Este aplica evolución diferencial como método de optimización, utilizando únicamente el vector de probabilidades de la salida. Gracias a esto es posible usar este ataque tanto en modalidad *black-box* como en casos donde el modelo atacado no es diferenciable.

Feature Adversary

En este ataque de modalidad *white-box* los autores proponen generar una perturbación haciendo uso de dos imágenes, una imagen fuente y una imagen guía [29]. El método consiste en modificar la imagen fuente haciendo que la salida de una de las capas ocultas de la red se asemeje a la salida en la misma capa para la imagen guía.

De esta forma (y sin utilizar la predicción final de la red) se logra que la representación interna de la imagen fuente se asemeje a la de la guía, lo que en la mayoría de los casos produce un cambio en la clase final asignada por el clasificador.

Hot/Cold

Como mencionamos en el capítulo 2, la mayoría de los algoritmos utilizan las normas L_p como medida de similitud entre imágenes. Los autores de este ataque por otro lado proponen una nueva métrica denominada PASS que busca ser una mejor aproximación a la percepción humana [30].

Para esto tiene en cuenta factores como rotaciones o traslaciones de las imágenes comparadas, las cuales podrían corresponderse a distintos puntos de vista del mismo objeto y aún así ser consideradas como imágenes totalmente distintas bajo las normas L_p. Finalmente proponen el método iterativo *Hot/Cold* que utiliza PASS para generar ejemplos adversarios visualmente similares a la imagen original.

En la tabla 3.3 incluimos un resumen de los distintos ataques presentados, indicando para cada uno de ellos a qué categorías pertenecen de entre las nombradas en la sección 3.3, además de las métrica de distancia para las cuales están optimizados.

3.5. Tipos de Defensas

En la actualidad existe una creciente cantidad de soluciones basadas en aprendizaje automático que con frecuencia son aplicadas a problemas de carácter crítico o sensible, por lo que tras el descubrimiento de los ejemplos adversarios surge la necesidad de desarrollar

	Objetivo	Alcance	Medio del ataque	Conocimiento del atacante	Costo computacional	Métrica de distancia
L-BFGS	Dirigido	Específico	Digital	*white-box*	Iterativo	L_2
FGSM	No Dirigido	Específico	Digital	*white-box*	Un paso	L_{inf}
R+FGSM	No Dirigido	Específico	Digital	*white-box*	Un paso	L_{inf}
BIM	No Dirigido	Específico	Digital	*white-box*	Iterativo	L_{inf}
MI-FGSM	No Dirigido	Específico	Digital	*white-box*	Iterativo	L_{inf}
JSMA	Dirigido	Específico	Digital	*white-box*	Iterativo	L_0
DeepFool	No Dirigido	Específico	Digital	*white-box*	Iterativo	L_2
Universal Perturbations	No Dirigido	Universal	Digital	*white-box*	Iterativo	L_p
Carlini&Wagner	Dirigido	Específico	Digital	*white-box*	Iterativo	L_0, L_2, L_{inf}
EOT	Dirigido	Específico	Físico	*white-box*	Iterativo	L_p
BPDA	Dirigido	Específico	Digital	*white-box*	Iterativo	L_2
ZOO	Dirigido	Específico	Digital	*white-box*	Iterativo	L_2
Adversarial patch	Dirigido	Universal	Físico	*black-box*	Iterativo	L_2
One Pixel Attack	Dirigido	Específico	Digital	*black-box*	Iterativo	L_0
Feature Adversary	Dirigido	Específico	Digital	*white-box*	Iterativo	L_{inf}
Hot/Cold	Dirigido	Específico	Digital	*white-box*	Iterativo	PASS

Cuadro 3.2: Clasificación de los distintos ataques estudiados utilizando los distintos criterios planteados en la sección 3.3

sistemas robustos ante este tipo de ataques. Intuitivamente deseamos que si una red es capaz de procesar correctamente determinado ejemplo de entrada, también logre procesar de forma correcta ejemplos que son imperceptiblemente diferentes.

En este contexto entendemos por **defensa** a cualquier mecanismo que busque mejorar la robustez de una red volviéndola más resistente a ejemplos adversarios, y que, de ser posible, lo haga sin deteriorar su desempeño sobre ejemplos limpios. En el caso estudiado esto implica lograr que la clasificación $C(x')$ para imágenes alteradas x' sea igual a la clasificación $C(x)$ de sus imágenes originales, donde además x' y x son cercanas sobre alguna métrica de distancia d.

En conjunto a los distintos algoritmos para generar ejemplos adversarios también se desarrollaron diversos mecanismos de defensa que buscan mitigar sus efectos. En la próxima sección presentamos una posible clasificación para estas defensas, además de algunas de sus principales características. Posteriormente describimos una serie de defensas populares y culminamos su estudio comentando algunas consideraciones relevantes.

3.5.1. Clasificación

Las defensas, al igual que los ataques, pueden ser clasificadas de varias maneras según sus distintas características, siendo una de las más relevantes el impacto requerido sobre la red a defender. Esto da lugar a dos grandes grupos, el de las defensas **agnósticas al modelo** y el de las defensas **específicas al modelo**.

1. **Agnósticas al modelo**: Estas defensas no requieren de un conocimiento previo de la red a defender, sino que buscan mejorar su rendimiento operando únicamente sobre los datos de entrada. Se trata de algoritmos que de alguna forma intentan eliminar las posibles perturbaciones o propiedades no deseables presentes en los datos de entrada para que la red sea capaz de clasificarlos correctamente.

   El hecho de que no requieran cambios en el modelo protegido es una de sus principales ventajas, ya que esto en general implica que pueden ser aplicadas de forma rápida sobre redes ya existentes, sin necesidad de reentrenarlas o de realizar cambios a su arquitectura, siendo posible además usarlas junto con otros mecanismos.

   Dentro de estas defensas podemos identificar a su vez dos tipos de algoritmos:

   - **Detección**: Buscan determinar si los ejemplos recibidos consisten en datos **limpios** o por contrario de ejemplos adversarios. Entre los mecanismos para detec-

tar ejemplos adversarios se encuentran los basados en métodos estadísticos, los que entrenan un nuevo modelo detector y los que buscan inconsistencias de predicción [31]. La defensa puede decidir luego si rechazar el ejemplo recibido o hacer uso de alguna otra técnica de clasificación, teniendo en cuenta que se trata de una entrada perturbada.

- **Preprocesamiento**: Estos realizan alguna transformación sobre los datos de entrada previo a la clasificación de la red, con la cual se busca mitigar los efectos de las perturbaciones en caso de tratarse de ejemplos adversarios. Las técnicas utilizadas en este procesamiento están muy ligadas al tipo de datos con los cuales se esté trabajando, lo que las hace específicas al dominio y difíciles de aplicar en otras tareas. Por ejemplo, el procesamiento que se puede realizar sobre imágenes (los valores de sus píxeles) no es el mismo al que se puede aplicar a muestras de audio.

2. **Específicas al modelo**: Por otro lado, las defensas específicas al modelo buscan mejorar la robustez alterando los propios modelos para hacerlos menos susceptibles a los ejemplos adversarios. Se consideran a este tipo de defensas como *proactivas* dado que buscan mitigar el impacto de los ejemplos desde inicio del diseño, ya sea alterando su arquitectura o su método de entrenamiento, lo que nuevamente da lugar a dos grupos.

   - **Entrenamiento**: Son las técnicas que consisten en utilizar ejemplos adversarios durante la etapa de entrenamiento. El uso de estos ejemplos puede hacerse de forma parcial o total, utilizándolos en conjunto a imágenes limpias ó entrenando únicamente sobre imágenes adversarias.

   - **Arquitectura**: En estas se realizan modificaciones a la arquitectura de la red o se buscan nuevos métodos de entrenamiento que logren eliminar la sensibilidad ante ejemplos adversarios. Un enfoque simple sería aumentar la capacidad del modelo para lograr mejor generalización, mientras que otras defensas agregan técnicas de regularización.

Por otro lado existe otro tipo de defensas que ha sido muy adoptado en la actualidad, que pueden a su vez pertenecer a cualquiera de los dos grupos ya mencionados, y se caracterizan por inutilizar los gradientes de las redes. Estas reciben el nombre de defensas por **enmascaramiento de gradientes** y se basan en hacer que los gradientes obtenidos de las redes no existan o no aporten información de utilidad para los atacantes, ya sea mediante decisiones de arquitectura, algoritmos de entrenamiento o transformaciones aplicadas en la entrada.

Debido a que la mayoría de ataques en modalidad *white-box* hacen uso de los gradientes, este enmascaramiento logra efectivamente dificultar la búsqueda de ejemplos adversarios. Entre las distintas técnicas utilizadas para enmascarar el gradiente se encuentran las presentadas en la sección 3.4.8.

3.6. Defensas destacadas

En esta sección presentamos algunos de los mecanismos de defensa más populares para problemas de clasificación de imágenes. Al igual que en el caso de los ataques, nuestro objetivo es el de presentar de forma simplificada los distintos enfoques tomados en el abordaje de este problema, mencionando sus particularidades y motivaciones.

3.6.1. Input Transformations

Comenzamos por presentar un conjunto de mecanismos de defensa agnósticos al modelo, más precisamente del tipo de preprocesamiento. En [32] los autores presentan cinco técnicas sencillas que transforman las imágenes de entrada buscando eliminar los efectos adversarios de las perturbaciones. Como veremos a continuación, estos métodos utilizan tanto operaciones no diferenciables como aspectos de aleatoriedad por lo que también pertenecen al grupo de técnicas de enmascaramiento de gradiente.

- ***Crop and Rescale***: Esta consiste en recortar aleatoriamente múltiples porciones de cada imagen de entrada y posteriormente reescalarlas al tamaño de entrada original. La predicción final de la defensa se calcula luego como el promedio de predicciones para los distintos recortes realizados [4].

  Al aplicar estas transformaciones se modifica el posicionamiento espacial de las perturbaciones, contrarrestando así en gran medida su capacidad para atacar al modelo.

- **Bit depth reduction**: Busca eliminar pequeñas perturbaciones de las imágenes mediante una reducción en la cantidad de bits utilizados para codificarla, de 8 a únicamente 3 en los experimentos reportados.

- **Compresión JPEG**: JPEG es un método de compresión con pérdida para imágenes. Al comprimir las imágenes de entrada con este método se logran eliminar pequeñas perturbaciones de forma similar a el método anterior.

- ***Total Variance Minimization* (TVM)**: TVM es un método de procesamiento de imágenes para la eliminación de ruido. En este se hace una selección aleatoria de píxeles provenientes de la imagen de entrada y luego se genera una imagen nueva (similar a la original) partiendo del conjunto de píxeles seleccionado. Esta reconstrucción se hace buscando minimizar la *total variation*[5] de la imagen resultante, de forma que se tienden a eliminar pequeñas perturbaciones que causan efectos adversarios.

- ***Image Quilting***: Esta es una técnica que consiste en reconstruir las imágenes de entrada utilizando muchas porciones de imágenes limpias. Para esto se parte de un conjunto de parches pequeños (por ejemplo $5x5$ píxeles) obtenidos de forma aleatoria de un conjunto de imágenes limpias.

  Luego, para cada porción de las imágenes de entrada el algoritmo busca los parches que más se le asemejen y selecciona uno aleatoriamente, reemplazando así todos los píxeles de la imagen. Finalmente se manipulan las uniones entre parches para eliminar posibles irregularidades que puedan haberse formado.

  Dado que todos los píxeles son sustituidos con parches provenientes de imágenes limpias, se reduce considerablemente la probabilidad de que se mantengan las estructuras presentes en los ejemplos adversarios.

Los autores de [32] evalúan estas técnicas en varios escenarios y contra varios algoritmos de ataque populares (FGSM, BIM, DeepFool y Carlini&Wagner), donde el atacante no tiene conocimiento del modelo atacado (modalidad *black-box*) así como en el caso que el atacante tiene conocimiento del modelo pero no del mecanismo de defensa, además de los casos en el cuales las transformaciones son aplicadas únicamente en tiempo de evaluación así como también en el cual son usadas durante el entrenamiento de las redes.

[4] En el trabajo original realizan 30 recortes de 90x90 píxeles por cada imagen de entrada, las cuales tienen un tamaño 224x224 provenientes del *dataset* ImageNet.

[5] Métrica que refleja la cantidad de variación de una imagen considerando píxeles adyacentes.

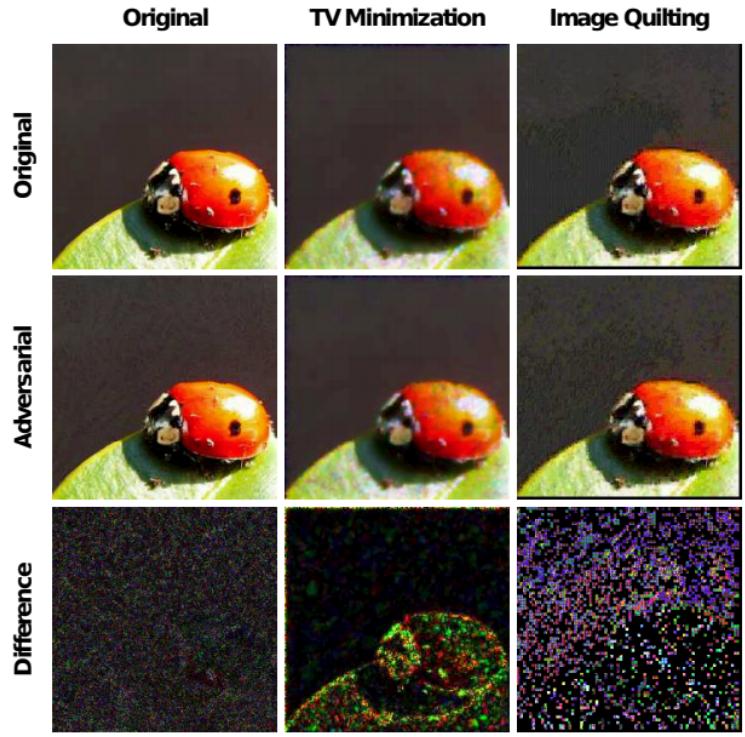


Figura 3.6: Muestra de la aplicación de TVM e Image Quilting sobre la imagen original y su adversaria (obtenida con I-FGSM)

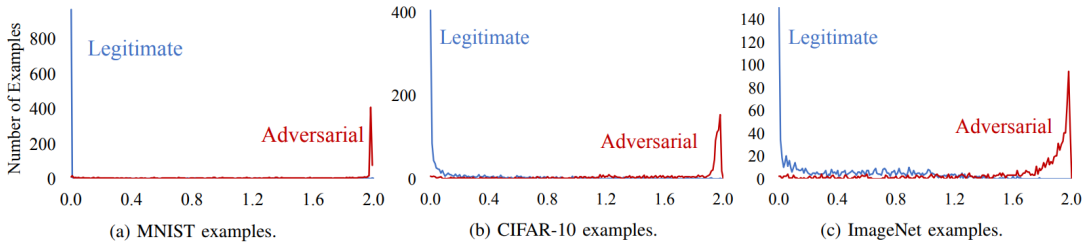
De los resultados obtenidos se deduce que estas técnicas son capaces eliminar parcialmente los efectos de los ejemplos adversarios, siendo la de *Crop and Rescale* una de las mejores al clasificar correctamente entre un 40 % y 60 % de los ejemplos. De todas formas, en el trabajo del ataque presentado en 3.4.8 se proponen técnicas para invalidar cada una de las defensas mencionadas previamente, logrando reducir su efectividad hasta el 0 %, por lo que estas no pueden ser consideradas como seguras en los casos en los cuales el atacante las conoce.

3.6.2. Feature Squeezing

Esta es otra técnica agnóstica al modelo, que en este caso busca detectar ejemplos adversarios en tiempo de evaluación. Para esto se apoya en la idea de que mediante el uso de transformaciones (como las presentadas en la defensa anterior) es posible eliminar características de las imágenes que son poco relevantes para su correcta clasificación, pero que sin embargo tienen un rol importante en la efectividad de los ejemplos adversarios. Los autores luego demuestran de forma empírica que las predicciones[6] de la red para imágenes limpias, obtenidas antes y después de aplicar las transformaciones no cambian de forma significativa. Sin embargo la diferencia de predicciones es considerable cuando se aplican dichas transformaciones sobre ejemplos adversarios.

Las transformaciones consideradas por los autores son *bit depth reduction* (presentada en 3.6.1) y filtrado de mediana (*median blur*). Proponen medir la distancia (con norma L_1) entre los vectores de predicción de la imagen original (con o sin perturbación adversaria)

[6]Concretamente, el vector de predicciones resultante para todas las clases consideradas.



(a) MNIST examples. (b) CIFAR-10 examples. (c) ImageNet examples.

Figura 3.7: Diferencia (en norma L_1) entre la predicción realizada sobre la imagen original y la predicción realizada sobre la imagen después de aplicar las transformaciones *squeeze* definidas por la defensa. En azul se indica esta diferencia para las imágenes legitimas mientras que en rojo la diferencia para imágenes con perturbaciones adversarias, destacando que se pueden diferenciar ambos casos claramente.

y compararlas con las predicciones luego de transformadas. Al elegir en cada ejemplo la transformación que presente mayor diferencia se obtienen los valores presentados en la figura 3.7, donde se observa claramente que los ejemplos adversarios tienden a presentar una diferencia alta entre sus predicciones, mientras que los ejemplos limpios tienen una diferencia cercana a 0 en la mayoría de los casos.

La defensa propuesta consiste entonces en definir un umbral adecuado de forma que los ejemplos que presenten una diferencia de predicción mayor a este valor sean considerados como ejemplos adversarios. Este método logra detectar ejemplos adversarios en un alto porcentaje de casos, variando entre el 85 % y 98 % para los distintos *datasets*.

3.6.3. PixelDefend

En esta defensa [31] los autores demuestran empíricamente que los ejemplos pertenecen a zonas de baja probabilidad dentro de la distribución de datos de entrenamiento, siendo este el principal motivo por el cual son clasificados de forma incorrecta.

Considerando esto proponen utilizar PixelCNN[7] para aprender la distribución de las imágenes de entrenamiento y de esa forma poder aproximar la probabilidad de que un ejemplo nuevo pertenezca a dicha distribución. Luego la defensa consiste en *purificar* las imágenes de entrada llevándolas hacia la misma distribución aprendida, tarea que resuelven buscando en la vecindad de x por un punto $x*$ que maximice la probabilidad $p(x*)$ estimada por la PixelCNN.

$$\begin{aligned} \max_{x'} \quad & p(x*) \\ \text{sa.} \quad & \|x - x*\|_\infty \leq \epsilon_{defend} \end{aligned}$$

De esta forma obtienen una defensa agnóstica al modelo que trata de determinar para cada imagen de entrada si se trata de un ejemplo adversario y en caso positivo purificarla de forma que pueda ser clasificada correctamente por el modelo original. Además, debido a que se basa en la propia distribución de las imágenes para eliminar las perturbaciones, también resulta agnóstica a los métodos de ataque utilizados.

[7] Modelo generativo que alcanza estado del arte en el modelado de distribuciones de imágenes.

3.6.4. Defense-GAN

Esta es aún otra defensa agnóstica al modelo que de forma similar a PixelDefend busca eliminar posibles perturbaciones adversarias proyectando los ejemplos de entrada hacia la distribución de los datos de entrenamiento. A diferencia del método anterior, este hace uso de una *Generative Adversarial Network* (GAN)[8] para aprender la distribución de imágenes de entrenamiento y posteriormente generar ejemplos no adversarios.

Una vez entrenada la GAN sobre el mismo *dataset* que el clasificador, el algoritmo de defensa consiste en utilizarla para buscar imágenes x^* lo más cercanas a las imágenes originales x. Sea G el modelo generador de la GAN y z una semilla de entrada, se utiliza descenso por gradiente para encontrar el z^* que minimice la siguiente ecuación:

$$\min_z \|G(z) - x\|_2^2$$

La reconstrucción $G(z^*)$ es finalmente usada como entrada para el modelo de clasificación y dado a que esta se encuentra en el rango del generador (entrenado sobre imágenes limpias) logra eliminar en gran medida los efectos de las perturbaciones adversarias. Con esto obtienen buenos resultados en los dos *datasets* evaluados por los autores, MNIST y F-MNIST[9], aunque dicha efectividad depende de forma directa de la expresividad alcanzada por la GAN, siendo su entrenamiento una tarea sumamente compleja que puede dificultar la correcta implementación de esta defensa.

Vale destacar que esta es la única de las defensa evaluadas en [26] a la cual los autores no logran invalidar completamente. Argumentan que si bien existen ejemplos adversarios dentro del rango del generador G, que por lo tanto no deberían modificarse al ser recibidos como entrada, el hecho de que se use descenso por gradiente y no un proyector perfecto hace que dichos ejemplos no se preserven, limitando la efectividad del ataque.

3.6.5. Entrenamiento adversario

Desde los primeros estudios realizados sobre ejemplos adversarios una de las principales técnicas propuestas para mejorar la robustez de las redes ha sido la de agregar dichos ejemplos al conjunto de datos de entrenamiento. Esto se volvió más fácil de llevar a la práctica tras la aparición de métodos como el FGSM, el cual es capaz de generar ejemplos adversarios de forma más eficiente. Para este sus autores proponen entrenar una red utilizando la siguiente función de **pérdida adversaria**[10]:

$$\tilde{J} = \alpha J(\theta, x, y) + (1 - \alpha) J(\theta, x + \epsilon \operatorname{sign}(\nabla_x J(\theta, x, y)), y)$$

siendo α el parámetro que regula la penalización adversaria.

El resultado es una defensa específica al modelo, que además supone otras ventajas para la red defendida. Se corrobora que el entrenamiento adversario funciona como una técnica

[8] Las GANs [33] son un tipo de redes generativas capaces de generar ejemplos nuevos pertenecientes a una distribución de datos dada. Estas hacen uso de dos modelos, generador y discriminador, que son entrenados simultáneamente. El modelo generador aprende a generar ejemplos de la distribución de entrenamiento mientras que el discriminador aprende a distinguir entre ejemplos reales y ejemplos falsos (generados artificialmente).

[9] *Dataset* con prendas de ropa de iguales características que MNIST en cuanto a cantidad de clases y ejemplos, así como dimensión de las imágenes.

[10] La función de costo (o error) adversaria es como la función de costo tradicional, pero en vez de considerar simplemente la pérdida para cada dato de entrada, se considera el peor caso de pérdida en una región cerca de la entrada [34].

más de regularización, previniendo *overfitting* sobre los datos de entrenamiento y ayudando a la generalización de los modelos. También se puede ver como una forma de *data augmentation*,[11] que se diferencia a la técnica de aumento tradicional dado que en esta se utilizan ejemplos que no ocurren naturalmente en el corpus, sino que son datos que exponen fallas en cómo el modelo genera su función de decisión [5].

3.6.6. Entrenamiento adversario robusto

A diferencia del enfoque anterior, el entrenamiento adversario robusto propone utilizar solo entradas adversarias en el entrenamiento del modelo, siguiendo la idea de entrenar con el "peor" caso posible. Se presenta el problema de optimización que se intenta resolver en el entrenamiento como una fórmula *min-max*, donde se quiere minimizar la función de costo del modelo con respecto a una perturbación que este generando la máxima pérdida posible. En otras palabras, se buscan los parámetros del modelo que minimicen una función de pérdida adversaria.

Determinando la función de pérdida adversaria, la perturbación generada para cada entrada se encuentra dentro de un conjunto $S \subseteq R^d$ de perturbaciones posibles para el dato x, el cual define el poder del atacante en la manipulación de los datos. En clasificación de imágenes, es deseable que S represente perturbaciones que mantengan una similitud perceptual entre la imagen original y el ejemplo adversario. Pero hay que tener muy presente en cómo se quiere formar este conjunto S ya que dependiendo que tan compleja sea la técnica utilizada para crear las perturbaciones se impacta directamente en el tiempo necesario de entrenamiento, pudiendo aumentar considerablemente. Una opción es crear perturbaciones basándonos en la norma L_p por su sencillez en cálculo y una distorsión poco perceptible. Entonces, un ejemplo para el conjunto S serían las perturbaciones que caen dentro de $B_p(x, r)$ la bola de norma p centrada en x con radio r ó $\|\cdot\|_p$.

Dentro de este tipo de formulación de defensa, se encuentra la propuesta por Madry [35], donde utiliza descenso por gradientes proyectado (PGD) como mecanismo para encontrar los parámetros de la red de forma que ésta sea robusta a la peor perturbación (o el problema de maximización interno).

$$\min_{\theta} \rho(\theta, D_{train}) \equiv \min_{\theta} \frac{1}{|D_{train}|} \sum_{(x,y)\in D_{train}} \max_{\delta\in S} J(\theta, x+\delta, y)]$$

Lo más importante que sugieren con la experimentación sus autores es que este tipo de entrenamiento es robusto contra los ataques de primer orden, es decir, contra los ataques que se basan en la información del gradiente (o primera derivada) para encontrar la perturbación adecuada.

3.6.7. Conjunto de entrenamiento adversario

Como una alternativa más a las defensas específicas al modelo por entrenamiento se tiene la presentada por [20], que consiste en entrenar redes adversariamente pero incluyendo ejemplos perturbados que son generados a partir de otros modelos preentrenados previamente. Los autores se basan en la propiedad de transferibilidad de los ejemplos adversarios, explorando la idea de que utilizar ejemplos generados de otros modelos debe ayudar a la

[11] Técnica comúnmente utilizada para aumentar el tamaño de los datasets de entrenamiento cuando se cuenta con una cantidad limitada de ejemplos para el problema a resolver. Generalmente se trata de transformaciones, como traslación o rotación, sobre el conjunto de datos disponible.

robustez cruzada entre los mismos. A diferencia del enfoque en las defensas por entrenamiento anteriores, aquí se separa la generación de gran parte de los ejemplos utilizados en el entrenamiento del modelo que se está queriendo entrenar, además de que se aumenta la diversidad de ejemplos adversarios generados.

3.6.8. Destilación de redes neuronales

Dentro del área del aprendizaje profundo, se utiliza la técnica de destilación para transferir conocimiento de una red neuronal a otra de menor tamaño. Con esto se logra reducir la complejidad computacional mientras que se mantiene el poder descriptivo del modelo original.

Para conseguirlo se entrena al modelo más pequeño referido como "estudiante" con etiquetas suavizadas que se obtienen del modelo original al que se refiere como "maestro". Estas nuevas etiquetas son la última capa del modelo original o la capa *softmax* que representa la distribución de probabilidades para cada clase, por lo que contienen más información que las etiquetas *one-hot-encoded* comúnmente utilizadas.

En la función softmax utilizada, tanto para obtener las predicciones del modelo maestro como en el entrenamiento del estudiante, se introduce un parámetro de temperatura T de la siguiente manera:

$$F(\theta, x) = softmax(Z(\theta, x)/T)$$

El valor de este parámetro T, afecta la distribución de probabilidades de las clases resultantes. Cuanto $T \to \infty$, la distribución tiende a ser uniforme, por eso en la destilación de conocimiento, se utiliza un valor de T alto[12] obteniendo así las etiquetas mas suavizadas.

Utilizando esta técnica, Papernot [36] propone destilar la red ya entrenada a otra con la misma arquitectura e hiperparámetros logrando el efecto de reducir la amplitud de sus gradientes. Esto ayuda a combatir los ejemplos adversarios dado que el tener un rango menor de los gradientes implica que pequeñas perturbaciones en la entrada tienen menor variación en la salida.

Para terminar la sección de defensas, resumimos en el siguiente cuadro (3.5.1) las propuestas presentadas con su clasificación correspondiente.

	Tipo	Método
Input Transformations	Agnóstico al modelo	Preprocesamiento
Feature Squeezing	Agnóstico al modelo	Detección
PixelDefend	Agnóstico al modelo	Detección
DefenseGAN	Agnóstico al modelo	Preprocesamiento / Generativa
Entrenamiento Adversario	Específico al modelo	Entrenamiento
Entrenamiento Adversario Robusto	Específico al modelo	Entrenamiento
Conjunto de Entrenamiento Adversario	Específico al modelo	Entrenamiento
Destilación de Redes Neuronales	Específico al modelo	Arquitectura

Cuadro 3.3: Clasificación de las defensas presentadas según el criterio dado 3.5.1

[12] Por ejemplo, en el trabajo original experimentan hasta un máximo de $T = 100$

3.6.9. Comentarios

A pesar de la variedad de defensas propuestas hasta el momento, aún no existen mecanismos capaces de garantizar seguridad ante todos los ataques existentes, y el desarrollo de redes robustas contra ejemplos adversarios permanece como un problema abierto.

Históricamente el enfoque más utilizado para proponer defensas ha consistido en desarrollar métodos que son luego evaluados contra diversos ataques existentes. Esto de todas formas no es suficiente para determinar la robustez de una defensa, y dichos mecanismos suelen ser contrarrestados por ataques **adaptativos** que se desarrollan posteriormente aprovechando conocimiento sobre la defensa.

Propuestas más recientes se centran en obtener una verificación formal de la defensa, donde desde su construcción se certifique cierta robustez ante ataques bajo condiciones dadas. Igualmente, a pesar de que es la dirección correcta de investigación es un enfoque difícil de escalar para *datasets* de gran dimensión.

Dado que no es posible asegurar la robustez de una defensa en todos los escenarios posibles, resulta sumamente importante aclarar en cada caso bajo qué condiciones se argumenta seguridad, mediante el denominado modelo de amenaza. Esto incluye por ejemplo la norma máxima para las perturbaciones y el conocimiento que poseen los atacantes, tanto sobre la red misma como sobre los mecanismos de defensa aplicados. Una buena defensa no debería depender de un modelo de amenaza irreal que agregue restricciones exageradas como limitar la cantidad de consultas a la red, sino que debería ser capaz de garantizar la correctitud de las predicciones aún en los casos donde el atacante tiene un conocimiento completo del sistema atacado.

3.7. ¿Por qué existen los ejemplos adversarios?

A pesar del gran número de métodos desarrollados, tanto para generar como para defenderse de ejemplos adversarios, las causas detrás de su existencia aún no se conocen con seguridad.

Previo a describir algunas de las hipótesis planteadas, debemos presentar un concepto que se encuentra ligado y es usado comúnmente en el área del aprendizaje automático, el de **manifold** (variedad). En este contexto, el *manifold* refiere a conjuntos de puntos conectados de los cuales se puede obtener una buena aproximación utilizando un bajo número de dimensiones, dentro de un espacio de mayor dimensionalidad [37].

Esto es importante debido a que muchos algoritmos se basan en la **hipótesis del manifold**, que asume que la mayoría de los puntos del espacio consisten en datos de entrada inválidos, mientras que los puntos de interés pertenecen a distintos *manifolds*, los cuales a su vez contienen pequeños subconjuntos de puntos. Además de esto, se asume que las variaciones relevantes en la salida ocurren únicamente al desplazar las entradas dentro del *manifold* o al moverlas a *manifolds* diferentes.

Si bien estas suposiciones no resultan válidas en todos los casos, se cree que son bastante acertadas para tareas relacionadas al procesamiento de imágenes, audio y lenguaje natural. Debido a esto, es común hablar del *manifold* en el contexto de clasificación de imágenes para referirse a las regiones de $\mathbb{R}^n$ donde se encuentran la mayoría de las imágenes naturales o aquellas vistas durante el entrenamiento del modelo.

A continuación presentaremos algunas de las hipótesis que intentan explicar la existencia de los ejemplos adversarios.

Datos de entrenamiento incompletos

Una primera hipótesis de por qué existen los ejemplos adversarios sugiere que estos se corresponden a regiones de baja probabilidad del *manifold* de datos [4], las cuales no son visitadas durante la etapa de entrenamiento y por lo tanto resultan difíciles de clasificar correctamente.

Esto podría explicar el hecho de que los ejemplos generados para determinada red se transfieran con frecuencia a redes diferentes, siendo que los *datasets* de entrenamiento usados son en general los mismos, y aunque se utilicen subconjuntos disjuntos pertenecen a la misma distribución de datos. Estas regiones si bien no son fáciles de encontrar buscando puntos de forma aleatoria alrededor de la entrada, se pueden encontrar eficientemente mediante técnicas de optimización.

Linealidad

Los autores del ataque FGSM [5] en el mismo trabajo proponen otra hipótesis para la existencia de los ejemplos adversarios. Refutan la idea anterior de que estos se encuentran en pequeñas regiones del espacio y afirman que por lo contrario, ocupan grandes regiones que están relacionadas a la dirección de las perturbaciones más que a puntos específicos.

Para justificarlo argumentan que las redes neuronales se comportan de forma demasiado lineal debido a que las funciones de activación usadas comúnmente tienen a su vez un comportamiento muy lineal que las hace más fáciles de optimizar. Gracias a esto dicen que es posible engañar a los modelos realizando pequeñas modificaciones en los ejemplos de entrada, las cuales al tratarse de un espacio de muchas dimensiones logran acumularse para generar una gran diferencia en la salida.

Correlaciones en los límites de decisión

Tras demostrar la existencia de perturbaciones universales capaces de engañar a los modelos para múltiples ejemplos, los autores de *universal perturbations* [9] realizan un estudio intentando explicar este fenómeno, y concluyen que las perturbaciones explotan *correlaciones geométricas* en los límites de decisión del modelo. Para esto comparan el desempeño de sus perturbaciones con el de perturbaciones aleatorias de igual magnitud, donde observan que las generadas por el algoritmo propuesto obtienen resultados muy superiores al de las aleatorias, dando un indicio de lo planteado. Luego, apoyados por otros experimentos proponen la existencia de un subespacio $\mathcal{S}$ de menor dimensionalidad que el espacio de entradas, que contiene la mayoría de vectores normales a los límites de decisión en regiones rodeando a las imágenes naturales. Finalmente comentan que la efectividad de las perturbaciones universales se debe en parte a la existencia de este subespacio.

Geometría del manifold

En [38] se presenta un estudio de otra posible explicación para la existencia de los ejemplos adversarios, donde los autores intentan demostrar que este fenómeno se debe a la "alta

dimensionalidad geométrica de los *manifolds* de datos" en combinación con índices de error no nulos.

Definen un *dataset* sintético en el cual realizan una serie de experimentos, los cuales en conjunto a un análisis teórico del problema usan para demostrar que cuando el *manifold* tiene una alta dimensionalidad, aún las redes con un índice de error sumamente pequeño presentan ejemplos adversarios cerca de puntos clasificados correctamente.

Considerando a $d(E)$ como la distancia media (en L_2) desde entradas x al conjunto de puntos de error E, los autores logran demostrar sobre el *dataset* definido un límite superior para $d(E)$ en función del índice de error, y argumentan que para aumentar $d(E)$ de forma lineal, el índice de error debe disminuir significativamente. De todas formas resta estudiar en mayor detalle si resultados similares pueden demostrarse para *datasets* del mundo real.

En este capitulo presentamos los ataques adversarios, desde su definición a como pueden ser clasificados, junto a varios de los métodos destacados para generarlos. Para conocer como mitigarlos, presentamos algunas de las defensas más conocidas abarcando cada uno de los tipos posibles según la clasificación dada para las mismas. Finalmente mencionamos algunas de las hipótesis propuestas hasta el momento que intentan explicar los motivos detrás de este fenómeno.

En el próximo capítulo describimos el *framework* implementado, el cual busca simplificar la evaluación de algunos de los ataques y defensas presentados.

Capítulo 4

Framework para ataques adversarios

En conjunto al estudio teórico de los ejemplos adversarios nos planteamos el objetivo de experimentar sobre distintos algoritmos de ataques y defensas para obtener en consecuencia un mayor entendimiento en el área, al mismo tiempo de validar las afirmaciones realizadas por sus respectivos autores.

Con esto en mente desarrollamos un *framework* que permite incorporar los distintos algoritmos estudiados para posteriormente evaluarlos de forma sencilla y reproducible, obteniendo métricas de interés tales como el *accuracy* de las defensas ante distintos ataques, la norma de las perturbaciones generadas, etc. Todo esto se incluye dentro de un entorno el cual creemos puede resultar de utilidad también para otros investigadores, permitiéndoles centrarse en la implementación de sus algoritmos sin tener que preocuparse por una serie de factores que de otra forma consumirían tiempo de trabajo.

Este *framework* está orientado principalmente al uso en redes de clasificación de imágenes, aunque gracias a su diseño en el cual la mayor parte de la lógica se delega a una serie de componentes implementables por el usuario (como veremos a continuación), es posible extender su uso al estudio de ejemplos adversarios en otras áreas fuera de la visión artificial.

La figura 4.1 contiene una representación de la arquitectura diseñada. En esta podemos observar que la misma se encuentra dividida en dos grupos de componentes a los que denominamos **Componentes de usuario** y **Módulo de ejecución**.

El primero está compuesto por componentes que representan los distintos conceptos importantes dentro del estudio de ejemplos adversarios, como los mecanismos de ataque y defensa. Mediante estos componentes el usuario es capaz de incluir sus propios algoritmos, *datasets* y arquitecturas de redes para evaluarlos dentro del marco de ejecución. Dichos componentes son los siguientes:

- **Model**: Representa las redes neuronales.
- **DataSource**: Correspondiente a los *datasets*.
- **Defense**: Mecanismos de defensa.
- **Attack**: Algoritmos de ataque.
- **Task**: Métodos de evaluación e interacción con componentes.

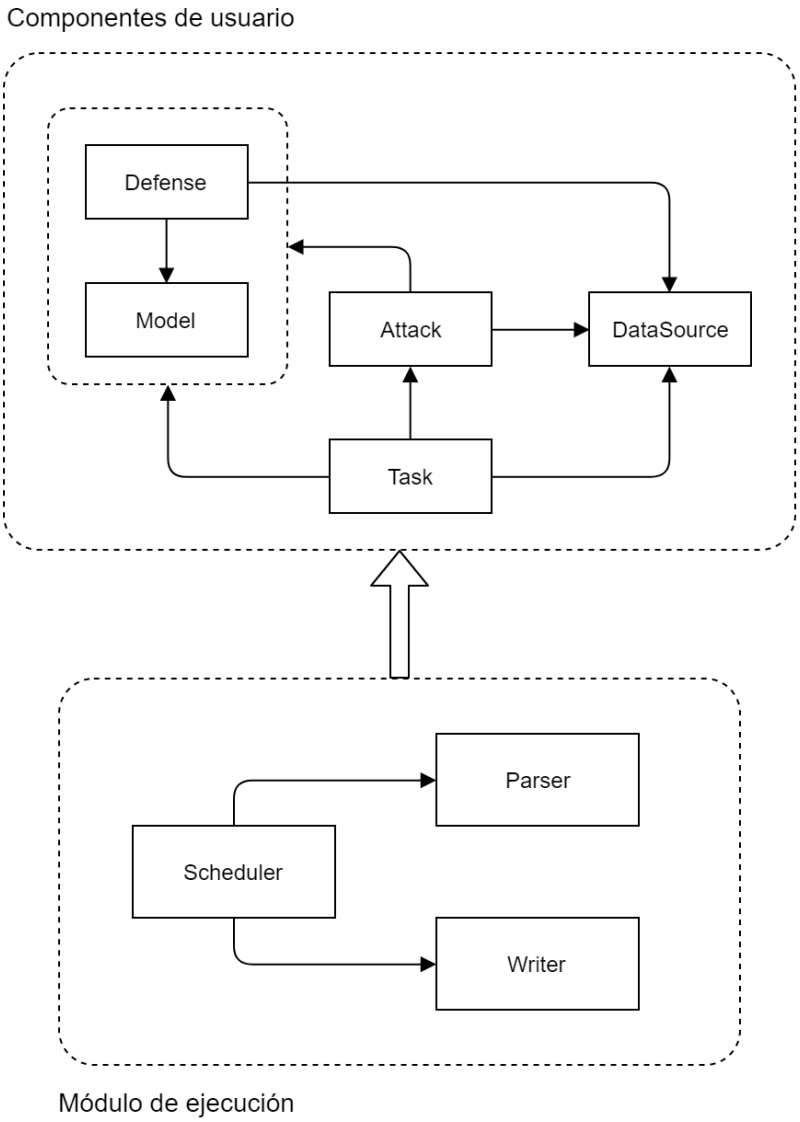


Figura 4.1: Arquitectura general del *framework* implementado.

Por otro lado el módulo de ejecución contiene los distintos componentes encargados del funcionamiento general del *framework*, y en principio no debe ser modificados por el usuario. Básicamente permite la especificación y ejecución de experimentos mediante los denominados **archivos de tareas**, que contienen una descripción de los distintos componentes de usuario a utilizar para obtener resultados reproducibles y fáciles de analizar.

En el resto de este capítulo profundizamos en los aspectos relevantes de la herramienta desarrollada. Comenzamos por presentar brevemente **PyTorch**, biblioteca de aprendizaje automático sobre la cual basamos nuestra implementación, para posteriormente centrarnos en los detalles de diseño y funcionamiento del *framework*.

4.1. Acerca de PyTorch

Antes de adentrarnos en los detalles de implementación del *framework* comenzamos por presentar la plataforma sobre la cual este está construido, **PyTorch**.

PyTorch es una de las varias bibliotecas existentes dedicadas mayormente al aprendizaje

automático. Está desarrollada sobre Python y tiene como principales objetivos el brindar un potente manejo de tensores[1] con aceleración en GPU, además de proporcionar una completa plataforma para el desarrollo e investigación de técnicas de aprendizaje automático [2].

Es considerado una opción particularmente atractiva entre investigadores debido a su versatilidad y rápida curva de aprendizaje. Además está fuertemente integrado con Python (uno de los lenguajes de programación más usados por la comunidad científica) y cuenta con una serie de funcionalidades que permiten desarrollar y utilizar redes neuronales de forma rápida y sencilla. Por estos y otros motivos elegimos utilizar PyTorch como base para nuestra implementación.

A continuación presentamos de forma breve algunas de las muchas herramientas incluidas en dicha biblioteca, enfocándonos en aquellas que resultan útiles para el funcionamiento del *framework* y los componentes de usuario.

4.1.1. Manejo de datos

Los datos son uno de los factores más importantes de cualquier algoritmo de aprendizaje automático, por lo que PyTorch incluye una serie de paquetes y clases orientadas a facilitar su uso e integración con los algoritmos implementados.

El paquete **torch.utils.data** contiene una serie de clases dedicadas principalmente a la obtención de dichos datos, entre las cuales hacemos uso de las siguientes dos:

- **Dataset**: Una clase abstracta que representa los distintos datasets utilizados por los algoritmos de aprendizaje automático. Todos los *datasets* deben heredar de esta clase e implementar un simple comportamiento que consiste en retornar ejemplos para los índices dados.
- **DataLoader**: Esta clase simplifica el manejo de los *datasets*, permitiendo utilizarlos como iterables y agregando algunas funcionalidades adicionales como el uso de lotes, la posibilidad de hacer *shuffle* sobre los datos para obtenerlos en orden aleatorio y utilizar múltiples procesos para acelerar la carga.

Otro de los paquetes que hacen de PyTorch una excelente herramienta para trabajar de forma rápida es el de **torchvision**. Este consiste en una serie de implementaciones de alto nivel orientadas a la visión artificial.

En cuanto al manejo de datos incluye los siguientes dos paquetes:

- **torchvision.transforms**: Este contiene una serie de transformaciones comunes de imágenes que simplifican la tarea de pre-procesamiento de los datos y permiten aplicar técnicas de *data-augmentation*. Las transformaciones disponibles incluyen reescalado, recorte y normalización de imágenes entre muchas más. Esto hace posible adaptar tanto el tamaño de las imágenes como sus rangos de valores para un uso óptimo.
- **torchvision.datasets**: En este encontramos implementaciones para los *datasets* de imágenes más populares, entre los cuales se encuentran todos los presentados en la sección 2.5. Estos heredan de la clase *Dataset* presentada anteriormente, por lo que es posible usarlos en conjunto a la clase *DataLoader*. Además presentan algunas

[1] Dentro del aprendizaje automático se suele usar el nombre **tensor** para referirse a arreglos multidimensionales.

[2] https://pytorch.org/

funcionalidades adicionales, entre las cuales se destacan el manejo de conjuntos de entrenamiento y de evaluación, descarga directa de los datos del *dataset* y la posibilidad de indicar *transforms* para aplicarse automáticamente a los ejemplos obtenidos.

4.1.2. Gradientes y Optimización

Como vimos en los capítulos anteriores, la optimización juega un papel central tanto en la generación de ejemplos adversarios como en los propios algoritmos de aprendizaje automático, donde la gran mayoría de estos se basan en el uso de gradientes para maximizar o minimizar determinadas funciones.

Una de las principales características de PyTorch es su capacidad de calcular gradientes de forma automática. Para esto basta con indicar qué tensores requerirán el cálculo de gradientes y PyTorch se encarga de generar un grafo dinámico registrando todas las operaciones realizadas sobre ellos, lo que posteriormente permite realizar el cálculo de los gradientes requeridos.

Adicionalmente, se incluye el paquete **torch.optim** que contiene la implementación de varios algoritmos de optimización, como descenso por gradiente estocástico, Adam [23], etc. En términos generales, estos toman el conjunto de parámetros a optimizar y se encargan de actualizarlos utilizando los gradientes calculados.

También con el objetivo de simplificar el entrenamiento de los modelos, dentro del paquete **torch.nn** se incluye un conjunto de funciones de costo como la *Cross Entropy Loss* usada comúnmente en los problemas de clasificación. El uso conjunto de estas herramientas hace que el entrenamiento de las redes y la optimización de funciones en general sean tareas relativamente sencillas y fáciles de implementar.

4.1.3. Modelos

También dentro del paquete **torch.nn** se encuentra la clase abstracta **Module**, utilizada para representar las distintas capas o módulos de las redes neuronales, así como algunas operaciones que llevan lugar dentro de estas. Estos *Modules* pueden ser anidados para generar estructuras más complejas, eventualmente formando redes neuronales completas las cuales también heredan de la clase *Module*.

Desde un punto de vista práctico, estos contienen la lógica necesaria para procesar datos de entrada y generar datos de salida mediante el uso de la función **forward** que debe ser implementada en cada caso. Para esto pueden utilizar un conjunto de parámetros o pesos como es el caso de los filtros en las capas de convolución.

Por medio de estos módulos PyTorch incluye una serie de implementaciones de capas usadas comúnmente en las redes neuronales, como las capas de convolución, las totalmente conectadas y las de *pooling*. Por otro lado, también como subclases de *Module* se incluyen algunas de las funciones de activación más populares como la sigmoide y la ReLU. Notar que a diferencia de las capas de convolución o las totalmente conectadas, las capas de *pooling* y las funciones de activación no poseen parámetros entrenables, por lo que su comportamiento no cambia luego de definidas.

Estas herramientas simplifican la tarea de modelado de las redes; sin embargo el paquete **torchvision** otorga una opción aún de más alto nivel para tareas de visión artificial. Dentro de **torchvision.models** se incluye una serie de redes populares utilizadas para tareas como clasificación, detección de objetos y segmentación semántica. Estas incluso

tienen la opción de obtenerse ya entrenadas, por lo que permiten a los usuarios comenzar a experimentar rápidamente sin necesidad de pasar por el proceso de diseño y entrenamiento de la red.

4.2. Componentes de Usuario

En esta sección introducimos los distintos componentes de usuario considerados en el *framework*. Mediante estos el investigador es capaz de incorporar sus propios algoritmos y criterios de evaluación para sus experimentos.

4.2.1. Model

Como vimos en capítulos anteriores, la existencia de ejemplos adversarios es una vulnerabilidad propia de las redes neuronales. Este componente es el encargado de representar las redes dentro del *framework*, permitiéndole al usuario especificar sus propias arquitecturas además de usar modelos ya existentes, e incluso ya entrenados. Dado que nos centramos en la clasificación de imágenes, tratamos en general con redes convolucionales como las presentadas en el capítulo 2.

Para la implementación de este componente decidimos usar directamente la clase **torch.nn.Module** de PyTorch, dado que las redes estudiadas en general no presentan ninguna particularidad al momento de trabajar con ejemplos adversarios que impida el uso de soluciones ya implementadas. El hecho de usar esta clase sin modificaciones nos permite además integrar redes ya existentes al *framework* de forma directa y facilita su uso a usuarios ya familiarizados con PyTorch.

4.2.2. DataSource

Otro aspecto a considerar en el estudio de los ejemplos adversarios son los *datasets*, que como vimos (para el caso de aprendizaje supervisado) no son más que conjuntos de datos etiquetados. En el *framework* hacemos uso de estos datos tanto durante la etapa de entrenamiento de los modelos como durante la etapa de evaluación, ambas pudiendo eventualmente verse afectadas por la intervención de ataques o defensas.

Para el manejo de datos dentro del *framework* agregamos la clase abstracta **DataSource**, una capa de abstracción sobre las clases *Dataset* y *DataLoader* ya proporcionadas por PyTorch. El objetivo de esto es adaptarlas ligeramente al uso esperado dentro del *framework*, agregando algunas funcionalidades extras y simplificando el proceso de incorporar nuevos *datasets*.

Entre las funcionalidades agregadas se destacan las siguientes:

- **Normalizado/desnormalizado**: Dentro del aprendizaje automático es una práctica habitual normalizar los datos de entrada con el objetivo de acelerar la convergencia durante el entrenamiento [39]. Como vimos en la sección 4.1, en los *datasets* proporcionados por *torchvision.datasets* es posible normalizar las imágenes de forma automática utilizando las funciones de *torchvision.transforms*. De todas formas, en ocasiones es deseable volver dichas imágenes a su rango original, por ejemplo para visualizarlas correctamente, o para aplicarles algún procesamiento especial y luego normalizarlas de forma manual. Con el objetivo de simplificar estas tareas, la clase *DataSource* provee funciones tanto para normalizar como para desnormalizar

las imágenes de un *dataset* haciendo uso de valores de media y desviación estándar proporcionados por el usuario.

- **Clamp**: Tanto los algoritmos de ataque como los de defensa pueden modificar los valores de las imágenes de entrada, ya sea para aumentar el error de predicción o para disminuirlo. Durante este proceso es posible que los valores de los píxeles escapen del rango válido para imágenes, generando así resultados incorrectos. Para evitar esto muchos algoritmos suelen aplicar **clamp**, acotando los valores de las imágenes resultantes de forma que los píxeles que ya se encuentren dentro del rango permitido permanezcan sin modificación y los que se encuentren por encima o por debajo se vean acotados. Si bien los datasets de imágenes suelen utilizar valores que van desde 0 a 1, estos cambian luego de ser normalizados resultando en cotas que varían entre los diferentes *datasets* e incluso entre los canales de un mismo *dataset*, dificultando la tarea de mantenerlos dentro del rango válido. Para solucionar este problema, implementamos la función *clamp* directamente en el *DataSource*, haciendo uso de los mismos valores de media y desviación estándar del ítem anterior para calcular los rangos correctos en cada caso.

4.2.3. Defense

Parte importante del estudio de los ejemplos adversarios se centra en obtener redes que sean menos susceptibles a esta amenaza. Este es el componente mediante el cual se pueden incluir estas técnicas al *framework*, que como vimos en la sección 3.5 pueden diferir sustancialmente en cuanto a su funcionamiento. Para contemplar todos estos algoritmos decidimos aquí también separarlas en los dos grupos propuestos: defensas agnósticas al modelo y defensas específicas al modelo.

Con el objetivo de dar soporte a ambos tipos de defensa y simplificar su uso decidimos hacer que estas se comporten de forma similar a las propias redes neuronales, lo que en nuestro caso de estudio implica que sean capaces de recibir imágenes como datos de entrada y devolver su clasificación como resultado.

Para esto agregamos la clase abstracta **Defense**, la cual a su vez hereda de *torch.nn.Module*, que como mencionamos en la sección 4.1.3 es utilizada por PyTorch para representar a las capas de las redes, las funciones de activación y a los propios modelos. Esto hace posible el utilizar a las redes normales (sin defender) y a las defensas de forma casi indistinta en el resto del *framework*, simplificando considerablemente la tarea del usuario al momento de implementar las *Tasks* y *Attacks* que veremos a continuación, permitiéndoles abstraerse de esta diferencia y utilizar a ambos componentes como simples instancias de la clase *Module*.

Al ser inicializadas, las defensas reciben la referencia al modelo a defender (instancia de la clase *Module*), así como al *DataSource* correspondiente. Luego exponen dos métodos, los cuales son usados mayormente por las *Tasks* implementadas.

- **process**: El primer método es propio a las defensas agnósticas al modelo y tiene como objetivo aplicar algún tipo de procesamiento a las imágenes de entrada para eliminar el efecto de posibles perturbaciones adversarias, por lo que a efectos prácticos recibe y devuelve imágenes. Este método debe ser implementado por el usuario si desea agregar una defensa del tipo mencionado.
- **forward**: El segundo método expuesto es el *forward*, proveniente de la clase *Module* de PyTorch. Como ya mencionamos este método es el encargado de procesar los datos de entrada y generar la salida correspondiente, siendo en este caso la clasificación.

Para simplificar la tarea del usuario, las defensas ya tienen una implementación por defecto para el método *forward* que asume un tipo de defensa agnóstica al modelo. Este se encarga de recibir las imágenes de entrada y procesarlas utilizando el método *process* para luego delegarlas a la red defendida y que esta se encargue de realizar la clasificación.

Las defensas específicas al modelo por otro lado suelen hacer uso del modelo defendido y no sólo de los datos de entrada. En estos casos, el usuario debe redefinir el método *forward* por completo, obteniendo total control sobre el proceso de clasificación. De esta manera tiene la posibilidad de modificar el modelo original o directamente sustituirlo por un modelo diferente, como en el caso de *defensive distillation* [36].

4.2.4. Attack

Como el nombre lo indica, este componente representa los distintos algoritmos de ataques a considerar. En estos se encuentra la lógica necesaria para convertir ejemplos de entrada naturales (proporcionados por el *DataSource* correspondiente) en ejemplos adversarios, los cuales posteriormente pueden ser procesados por las distintas redes.

Este es un componente conceptualmente simple para el cual la complejidad reside totalmente en la lógica implementada por el usuario. Al igual que el resto de los componentes se implementa mediante una clase abstracta, **Attack**, la cual recibe como argumentos una instancia de la clase *torch.nn.Module* correspondiente al modelo o defensa que debe atacar y otra al *DataSource* utilizado.

De esta forma es posible implementar ataques tanto de modalidad *white-box* como *black-box*, además de que la referencia al *DataSource* le permite hacer uso de sus funcionalidades mencionadas o eventualmente utilizar más ejemplos del *dataset* en caso que la lógica del algoritmo lo requiera.

4.2.5. Task

Finalmente, las *Tasks* contienen toda la lógica encargada de utilizar a los demás componentes para generar resultados de interés o interactuar con ellos de cualquier otra forma. Son el mecanismo mediante el cual llevar a cabo los diversos experimentos necesarios, desde entrenar a las redes con distintos *datasets* hasta evaluar el desempeño de determinados ataques y defensas.

El hecho de modelar esta lógica como otro componente de usuario brinda cierta flexibilidad adicional al *framework*. De esta forma los usuarios tienen la posibilidad de utilizar *Tasks* ya existentes para evaluar sus propios algoritmos, por ejemplo usando las mismas métricas que otros investigadores, y también en caso de ser necesario tienen la posibilidad de implementar *Tasks* específicas que se adecúen a sus necesidades, aprovechando el entorno de ejecución proporcionado por el *framework*.

Una restricción de este componente es que debe retornar resultados serializables, lo que da flexibilidad al permitir generar resultados estructurados, como listas, diccionarios, etc. al mismo tiempo que facilita su almacenamiento de forma organizada en el formato seleccionado, que en nuestro caso es *json*. De todas formas, también proporcionamos mecanismos para en caso de ser necesario permitirle a estas *Tasks* almacenar resultados en otros formatos, como imágenes, tablas, etc.

Para brindar una interfaz de uso sencilla sin perder la flexibilidad decidimos considerar dos modos de ejecución en las tareas, que, como veremos posteriormente, son utilizados

por el módulo de ejecución según lo especificado en el archivo de tareas. Estos modos se corresponden a dos funciones abstractas en la clase *Task*, de las cuales el usuario debe proporcionar la implementación para al menos una de ellas.

- **exec_task_simple**: Este función se corresponde al modo de ejecución simple y representa la noción general de tareas como la hemos presentado hasta el momento, permitiendo al usuario utilizar instancias de los otros componentes de forma directa para ejecutar la lógica deseada.

  Esta función recibe un total de tres argumentos sin contar la ruta. El primero consiste de una instancia de la clase *torch.nn.Module*, que nuevamente puede tratarse de un modelo o de una de las defensas implementadas. Luego recibe la instancia del *DataSource* correspondiente, y finalmente de forma opcional puede recibir una referencia a un ataque, instancia de la clase *Attack*.

  Con estos datos es posible implementar todas las tareas mencionadas previamente, abarcando desde la etapa de entrenamiento hasta la etapa de evaluación de las redes, y todo esto mediante el uso de componentes fácilmente reutilizables que simplifican ampliamente el trabajo necesario al momento de utilizar o evaluar nuevos algoritmos, modelos y *datasets*.

- **exec_task_multi**: Si bien la función *exec_task_simple* permite implementar una amplia variedad de tareas y de forma relativamente sencilla, tiene la limitación de que no contempla el uso de múltiples defensas y ataques en forma conjunta. Esto es de utilidad por ejemplo si se desean generar resultados más complejos como gráficas comparativas de ataques y defensas, o para cualquier lógica que requiera el uso de múltiples algoritmos de forma simultánea, como puede ser evaluar el desempeño de varias defensas combinadas.

  Para dar soporte a este tipo de tareas agregamos el denominado modo de ejecución múltiple. Esta función también recibe instancias de *torch.nn.Module* y *DataSource* (aunque en este caso se tratan exclusivamente de modelos y no defensas) y a diferencia de *exec_task_simple*, recibe iterables de las defensas y ataques especificados. Esto le brinda un mayor control al usuario, permitiéndole utilizar todos los algoritmos especificados de la forma que sea necesaria, aunque por supuesto con una mayor complejidad a la del caso anterior.

Además de los ya mencionados, las tareas presentan un tercer método, el **exec_attack_eval**. El objetivo de este método es permitirle al usuario evaluar de forma rápida el impacto que tienen determinados hiper-parámetros de los ataques sobre los resultado de las tareas, por ejemplo para estudiar el *accuracy* de un ataque como el FGSM sobre distintos valores de épsilon. A diferencia de los anteriores este ya posee una implementación base en la cual hace uso del *exec_task_simple*, invocándolo múltiples veces para distintos valores del hiper-parámetro seleccionado.

4.3. Módulo de Ejecución

Este módulo es el encargado del funcionamiento del *framework*, que como ya mencionamos consiste básicamente en la especificación y ejecución de tareas. Para esto el usuario debe indicar en los denominados **archivos de tareas** el conjunto de componentes que desea utilizar para le ejecución de cada experimento, desde las *Tasks* a usar (encargadas de generar los resultados finales) hasta los distintos componentes que se deseen evaluar y sus respectivos parámetros.

Una vez listo el archivo de tareas es posible comenzar con su ejecución, durante la cual el *framework*, o más específicamente el **módulo de ejecución**, se encarga de llevar a cabo las siguientes acciones:

1. **Leer el archivo de tareas**: En primer lugar se lee y valida el archivo de tareas. Esto implica determinar los distintos componentes a utilizar con sus respectivos parámetros, además de una serie de parámetros adicionales que regulan algunos aspectos de la ejecución.

2. **Ejecutar las tareas**: Luego de validado el archivo de tareas se pasa a la etapa de ejecución. Durante esta etapa el *Scheduler* se encarga principalmente de invocar a las *Tasks* que correspondan, pasándoles en cada caso los distintos componentes especificados. Recordar que la lógica capaz de generar los resultados de interés se encuentra dentro de las tareas, por lo que el *framework* nada más se encarga de orquestar su ejecución.

3. **Guardar los resultados**: Finalmente se guardan los resultados obtenidos de forma organizada. Estos se ubican en una estructura de directorios que refleja los distintos componentes involucrados durante la ejecución, permitiendo al usuario identificar y estudiar los datos generados en cada experimento de forma rápida y sencilla. Los archivos de resultados no solo contienen las salidas generadas por las tareas, sino que también contienen la información contextual de qué algoritmos fueron utilizados para generar dichos resultados y con qué conjuntos de parámetros.

Estos pasos se mapean de cierta forma a los tres componentes del módulo de ejecución, aunque a diferencia de como los presentamos aquí, no se llevan a cabo de forma totalmente secuencial sino que se realizan en una serie de ciclos según lo requerido para cada experimento.

En las próximas secciones profundizamos en los aspectos relevantes relacionados a la ejecución de experimentos. Comenzamos por detallar la estructura de los archivos de tareas y continuamos por presentar los tres componentes que forman este módulo.

4.3.1. Archivo de tareas

Los archivos de tareas son el mecanismo mediante el cual los usuarios pueden especificar experimentos para su posterior ejecución. Consisten en archivos de formato *json*[3] los cuales contienen todos los datos necesarios para llevar a cabo dichos experimentos y generar los resultados de interés. Estos incluyen la lista de componentes a utilizar, sus respectivos parámetros de ejecución y algunos parámetros de configuración adicionales.

Para simplificar el proceso de especificación de estos experimentos, aún para casos en los cuales se desean generar numerosos resultados, decidimos estructurar estos archivos de forma jerárquica con distintas secciones para cada uno de los componentes del *framework*

A continuación presentamos estas secciones nombrando algunas de sus principales características. Para una descripción completa de los parámetros admitidos en el archivo de tareas ver el anexo 7.1.

[3]JSON es un formato de texto utilizado para almacenar y comunicar datos estructurados. Tiene la ventaja de que es fácil de interpretar y escribir tanto por humanos como mediante programas, contando con soporte en todos los lenguajes populares.

- **Parámetros de configuración**: Parámetros opcionales relacionados a la ejecución del experimento, como la ruta en la cual almacenar los resultados, uso de tarjetas gráficas, etc.
- **Lista de tareas**: Las distintas tareas del experimento pero en un sentido ligeramente más amplio al manejado hasta el momento. Esto es, cada elemento de esta lista no solo contiene la información de una *Task* determinada, sino que además contiene la información de todos los elementos a utilizar en su ejecución.

  Las siguientes son las distintas secciones presentes en cada elemento de esta lista:

  - **Información de la tarea**: Esto incluye el identificador de la misma, los parámetros usados para inicializarla (propios a cada tarea) y finalmente una serie de parámetros utilizados para alterar el comportamiento del *framework*, como el modo de ejecución de las tareas (simple o múltiple) y si se deben graficar resultados entre otros.
  - **Modelos y DataSources**: Lista con los identificadores de los modelos a utilizar, sus respectivos *DataSources* y conjuntos de pesos. Estos tres elementos se especifican siempre de forma conjunta por lo que nos referiremos a ellos simplemente como **datos de la red**.

    El identificar los pesos de forma independiente al modelo mediante el uso del denominado *net_ id* nos permite manejar múltiples instancias de un mismo modelo, por ejemplo para evaluar distintas configuraciones o métodos de entrenamiento
  - **Defensas**: Lista opcional en la cual se especifican los identificadores de las defensas a evaluar con sus respectivos parámetros. Tener en cuenta que estas defensas luego serán aplicadas sobre cada uno de los modelos especificados en la sección de *Modelos y DataSources*.
  - **Ataques**: Lista opcional con los identificadores y parámetros de los ataques a utilizar. También es posible especificar algunos parámetros adicionales, siendo el más relevante de ellos el de **modelos específicos**: lista de redes igual a la de *Modelos y DataSources* la cual permite evaluar a los ataques en modalidad *black-box*, generando ejemplos sobre los modelos sustitutos especificados.
  - **Variables de ataques**: Como vimos en la sección anterior, las tareas poseen un método que permite evaluar el impacto de distintos hiper-parámetros sobre los resultados de los ataques. Esta última sección es la que le permite al usuario especificar dichos hiper-parámetros con sus respectivos valores, los cuales posteriormente serán utilizados por el módulo de ejecución sobre todos los ataques incluidos.

En el cuadro 4.1 podemos ver un ejemplo de un archivo de tareas simple. En dicho archivo se especifica una única tarea a ejecutar (*accuracy* en este caso) utilizando el modelo *resnet14* para el *dataset* MNIST, el ataque *FGSM* con un valor de epsilon de 0.2 y la defensa *JPEG compression*.

Estos archivos otorgan una amplia flexibilidad al momento de definir los experimentos, pero la gran cantidad de componentes y parámetros involucrados pueden dificultar tanto la tarea de especificación de los mismos como la de su interpretación. Por este motivo utilizamos una librería llamada **jsonschema** [4], la cual nos permite especificar de forma precisa el

[4] https://pypi.org/project/jsonschema/

```
{
  "tasks": [
    {
      "task_data": {
        "task_name": "accuracy"
      },
      "nets": [
        {
          "model_name": "resnet14_mnist",
          "datasource_name": "mnist",
          "net_id": "resnet14_mnist",
          "datasource_params": {
            "batch_size": 32
          }
        }
      ],
      "attacks": [
        {
          "attack_name": "fgsm",
          "attack_params": {
              "epsilon": 0.2
          }
        }
      ],
      "defenses": [
        {
          "defense_name": "jpeg_compression"
        }
      ]
    }
  ]
}
```

Cuadro 4.1: Ejemplo de un archivo de tareas.

esquema esperado para los archivos de tareas, y posteriormente utilizar esta especificación para validar que las entradas proporcionadas tengan el formato correcto.

Adicionalmente usamos este mismo esquema para documentar cada uno de los elementos incluidos y posteriormente presentárselos de forma intuitiva al usuario, incluyendo la estructura esperada además de una descripción de qué representa y para qué se usa cada uno de los parámetros.

Luego, para comenzar con la ejecución basta con indicar el archivo de tareas que se desea ejecutar y el *framework* se encarga de llevar a cabo las etapas mencionadas anteriormente, comenzando por la lectura, realizada mayormente por el *Parser*, siguiendo por la propia ejecución a cargo del *Scheduler* y finalizando con el almacenamiento de los resultados por medio del *Writer*.

4.3.2. Parser

Este componente se encarga de interpretar el archivo de tareas especificado por el usuario y proporcionar al resto del *framework* las instancias de los distintos componentes involucrados en cada experimento. Estas instancias son luego utilizadas por el *Scheduler* quien lleva a cabo la ejecución de las *Tasks* usando los métodos y argumentos correspondientes,

almacena los resultados, etc.

El *Parser* cuenta con las siguientes tres clases:

- **Parser**: La clase que efectivamente lee el archivo de tareas. Esta se encarga de interpretar cada uno de los componentes especificados, agregando valores por defecto a los parámetros no proporcionados por el usuario (principalmente en el caso de la configuración y las tareas) para finalmente retornar las instancias de los componentes correspondientes.
- **DefenseIterable**: Como su nombre indica, esta clase no es más que un iterable de defensas. Este recibe las listas completas de defensas especificadas en el archivo de tareas y genera un iterable de instancias de *Defense*. Para esto hace uso del *Parser*, quien procesa instancias individuales de los componentes. Estos iterables son usados tanto por el *Scheduler*, que como veremos a continuación debe iterar sobre los distintos componentes, como por las propias *Tasks* en los casos que implementan el modo de ejecución múltiple.
- **AttackIterable**: Este es otro iterable análogo al de las defensas pero con algunas particularidades propias. Como vimos en la especificación de ataques, es posible indicar modelos específicos para utilizar en los ataques. Debido a esto, los iterables de ataques no solo retornan una instancia por cada ataque especificado, sino que en caso de que corresponda también pueden retornar varias veces el mismo ataque, cambiando el modelo utilizado para cada iteración.

El uso de estos iterables además de abstraer parte de la lógica al resto de los componentes permite manejar los distintos ataques y defensas de forma sencilla y sin la necesidad de instanciarlos simultáneamente. Esto es particularmente importante considerando que tanto los ataques como las defensas pueden eventualmente instanciar modelos propios como parte de su funcionamiento, los cuales en caso de manejarse de forma inadecuada podrían suponer un consumo de recursos que limitase la capacidad de ejecución del *framework*.

4.3.3. Writer

Este componente se encarga de guardar los resultados generados en cada experimento, los cuales al igual que las entradas consisten de archivos de tipo *json* muy similares a los archivos de tareas.

Tanto al desarrollar algoritmos nuevos como al evaluar algoritmos ya existentes puede llegar a ser deseable o incluso necesario realizar múltiples experimentos complejos, que involucren distintas redes neuronales, con varios ataques y defensas, diferentes conjuntos de parámetros e incluso distintas versiones de algoritmos. La combinación de estos factores hace que rápidamente se generen grandes cantidades de datos, los cuales en caso de no manejarse adecuadamente pueden llegar a dificultar las tareas de análisis llevadas a cabo por el usuario.

Para evitar estos problemas y permitir identificar los datos deseados de forma rápida, el *Writer* se encarga de guardar los resultados organizadamente mediante el uso de la estructura representada en la figura 4.2. Además agrega cierta información contextual a dichos resultados, que ayuda tanto a interpretarlos de forma sencilla como a reproducirlos en caso de que sea necesario.

A continuación presentamos brevemente la estructura utilizada para almacenar los resultados generados.

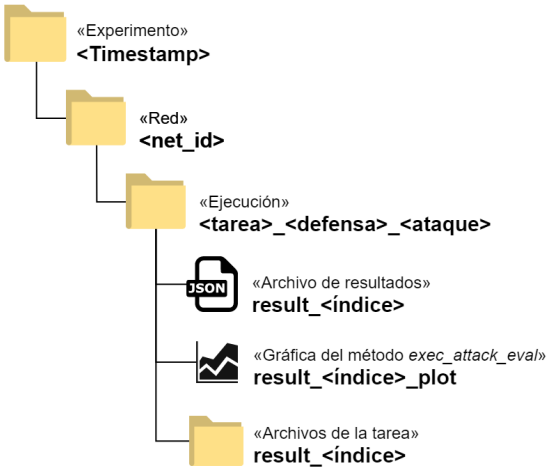


Figura 4.2: Estructura de directorios utilizada para almacenar los resultados de los experimentos.

- **Directorio del experimento**: En primer lugar, cada experimento realizado cuenta con un directorio propio, nombrado usando la fecha y hora actual. Esto permite separar los resultados de distintas ejecuciones además de ordenarlos cronológicamente.

- **Red**: El segundo nivel en la estructura se corresponde a la combinación de modelo, *DataSource* y pesos utilizada, y se nombra utilizando el *net_ id* correspondiente dado que este identifica de forma unívoca cada uno de los elementos mencionados. Mediante estos directorios se separan los resultados generados para las distintas redes usadas en cada experimento.

- **Tarea, defensa y ataque**: Este es el último nivel de directorios, aquí las carpetas se identifican utilizando el conjunto de componentes restantes. Cada una contiene el nombre de la tarea involucrada y en caso de haberlos el de la defensa y/o ataque del resultado. Esto le permiten al usuario identificar y acceder de forma rápida a todos los experimentos realizados sobre una red determinada.

- **Resultados**: Los resultados, como ya mencionamos consisten de archivos de formato *json* que se almacenan en el directorio anterior. Estos archivos contienen tanto los resultados generados por las tareas como la información contextual de su ejecución, compuesta por la porción del archivo de tareas correspondiente al resultado y los valores por defecto agregados por el *Parser*.

  En el caso de ejecutar la función *exec_ attack_ eval* de las tareas, es posible indicarle al *Writer* que también genere una gráfica con la lista de resultados obtenidos, permitiendo visualizarlos de forma más amigable.

  Finalmente, por cada archivo de resultados almacenado el *Writer* genera una carpeta adicional, cuya ruta es enviada a las *Tasks* al momento de ejecutarlas. De esta forma, los datos adicionales generados por cada tarea también se almacenan en el lugar adecuado y pueden ser vinculados al archivo *json* correspondiente.

El cuadro 4.2 contiene uno de los archivos de resultados generados para el archivo de tareas de ejemplo presentado en el cuadro 4.1. Se puede observar que la primera parte se asemeja a dicho archivo, en la cual se muestra la información de los componentes involucrados en esa ejecución además de algunos valores por defecto. Por último, a partir de la línea 36 se presentan los resultados tal cual fueron retornados por la *Task* en cuestión.

```
{
    "task_data": {
        "task_name": "accuracy",
        "task_params": {},
        "exec_multi": false,
        "attack_on_defense": true,
        "plot_results": false,
        "plot_keys": [],
        "plot_together": true,
        "skip_no_defense": false,
        "skip_no_attack": false,
        "skip_no_attack_variables": false
    },
    "net_data": {
        "model_name": "resnet14_mnist",
        "datasource_name": "mnist",
        "net_id": "resnet14_mnist",
        "datasource_params": {
            "batch_size": 32
        },
        "model_params": {}
    },
    "attack_data": {
        "attack_name": "fgsm",
        "attack_params": {
            "epsilon": 0.2
        },
        "except_variables": [],
        "on_task_model": true
    },
    "defense_data": {
        "defense_name": "jpeg_compression",
        "defense_params": {}
    },
    "exec_time": "0h 38m 19s",
    "result": {
        "total": 10000,
        "correct": 288,
        "adversarial": 9609,
        "accuracy": "2.88%",
        "correct_avg_confidence": "98.21%",
        "fooled_avg_confidence": "48.28%",
        "dataset_norm_0": 784,
        "dataset_avg_norm_2": 9.295949977111816,
        "dataset_avg_norm_inf": 0.9995996139526367,
        "adv_avg_norm_0": 478.39567072536164,
        "adv_avg_norm_2": 2.8324779876127786,
        "adv_avg_norm_inf": 0.13138700357795022,
    }
}
```


Cuadro 4.2: Ejemplo de un archivo de resultados generado por el *framework*.

4.3.4. Scheduler

Finalmente, el *Scheduler* se encarga de llevar a cabo el funcionamiento mismo del *framework*, interactuando tanto con los otros componentes del módulo de ejecución como

con las propias *Tasks*, para así obtener y almacenar los resultados de los experimentos especificados en los archivos de tareas. De todas formas este presenta una lógica propia relativamente simple, en la cual básicamente invoca a las distintas tareas utilizando todos los componentes que correspondan.

Recordando la estructura del archivo de tareas, vemos que por cada *Task* se especifican las listas de redes, ataques y defensas a evaluar. El *Scheduler* se encarga de iterar sobre todas sus combinaciones posibles, haciendo uso del *Parser* para obtener las instancias de cada componente. Una vez obtenidas, invoca el método correspondientes de la tarea y finalmente hace uso del *Writer* para almacenar los resultados obtenidos. Esto se repite hasta obtener todos los resultados del experimento.

Capítulo 5

Evaluación de ataques y defensas

En este capítulo presentamos los experimentos realizados y los resultados obtenidos. Mediante estos experimentos buscamos evaluar algunos de los ataques y defensas presentados en el capítulo 3, con el objetivo de corroborar las afirmaciones de sus respectivos autores al mismo tiempo de adquirir un mayor entendimiento en el tema y validar la utilidad del *framework* desarrollado.

A continuación describimos brevemente los componentes implementados para luego continuar con el análisis de los resultados obtenidos. Tanto el código del *framework* como los algoritmos implementados y los resultados presentados en este capítulo se encuentran disponibles en github[1].

5.1. Componentes implementados

Como vimos en el capítulo 4, el funcionamiento del *framework* gira en torno a una serie de componentes mediante los cuales el usuario puede incluir y evaluar sus propios algoritmos. Para verificar un correcto funcionamiento y experimentar con algunos de los algoritmos estudiados decidimos implementar una serie de ataques y defensas, así como un par de tareas, modelos y *datasets*.

Datasets

Para el manejo de los datos incluimos un *DataSource* por cada uno de los *datasets* presentados en la sección 2.5: **MNIST**, **CIFAR-10** e **ImageNet**.

Algo a tener en cuenta es que las imágenes de todos los *datasets* fueron normalizados para su uso. En el caso de ImageNet utilizamos los mismos valores de media y desviación estándar considerados durante el entrenamiento de los modelos seleccionados [40], mientras que para MNIST y CIFAR-10 utilizamos un valor de 0,5 para ambos casos. Esto es relevante debido a que al normalizar las imágenes se modifican los rangos de valores posibles dificultando la tarea de comparación con los resultados originales, los cuales suelen presentarse sobre imágenes en los rangos $[0, 1]$ o $[0, 255]$.

Teniendo en cuenta esto, los resultados aquí presentados fueron adaptados para corresponder a los rangos de los trabajos originales en cada caso.

[1] https://github.com/diegoirigaray/CrAdv

	Media	Desviación estándar
MNIST	0.5	0.5
CIFAR-10	[0.5, 0.5, 0.5]	[0.5, 0.5, 0.5]
ImageNet	[0.485, 0.456, 0.406]	[0.229, 0.224, 0.225]

Cuadro 5.1: Valores de media y desviación estándar utilizados para normalizar los distintos *datasets* considerados, de imágenes originalmente en el rango $[0, 1]$. Los casos con tres valores se corresponden a los distintos canales: rojo, verde y azul respectivamente.

Modelos

En cuanto a los modelos hicimos uso de varias arquitecturas populares para clasificación de imágenes: **ResNet** [41], **VGG**, **Inception V3** [42] y **DenseNet** [43], todas ellas logran obtener buenos resultados sobre imágenes limpias.

Para los casos de MNIST y CIFAR-10 diseñamos dos variantes a la red ResNet-18, a las que nos referiremos como ResNet-14_mnist y ResNet-14_cifar. Estas contienen la misma disposición de bloques que la ResNet-18 pero disminuyendo su tamaño con el objetivo de adaptarlas a la menor dimensión de imágenes de estos *datasets* (originalmente diseñada para ImageNet), para un total de 14 capas en lugar 18 y una menor cantidad de filtros de convolución.

Para ImageNet por otro lado utilizamos varias de las arquitecturas disponibles, en estos casos sin modificaciones: ResNet-50, VGG-19, Inception v3 y Densenet-121. Esto nos permite obtener los parámetros de dichas redes ya entrenadas, evitándonos la demandante tarea de entrenarlas nosotros mismos.

La siguiente tabla contiene los distintos valores de *accuracy* Top-1 alcanzados por los modelos utilizados, además del *accuracy* Top-5 para los modelos entrenados en ImageNet:

Modelo	Top-1	Top-5
ResNet-14_mnist	98,96 %	-
ResNet-14_cifar	86,96 %	-
ResNet-50	76,13 %	92,86 %
VGG-19	72,38 %	90,88 %
Densenet-121	74,65 %	92,17 %
Inception v3	77,45 %	93,56 %

Cuadro 5.2: *Accuracy* de los distintos modelos utilizados sobre imágenes limpias. A excepción de los modelos ResNet-14_mnist y ResNet-14_cifar, el resto de valores se corresponden a las redes pre-entrenadas del paquete *torchvision* de PyTorch [40].

Ataques

Para el caso de los ataques implementamos algunos de los métodos *white-box* más populares propuestos hasta el momento, todos de los cuales fueron presentados en el capítulo 3.

Estos son **FGSM**, **BIM**, **MI-FGSM**, **DeepFool** y **Carlini&Wagner** para norma L_2. Dichos métodos nos brindan cierta variedad en cuanto a complejidad y capacidad de ataque, variando desde métodos simples y rápidos como el FGSM hasta métodos más potentes y demandantes como el Carlini&Wagner.

Defensas

Para experimentar con las defensas implementamos dos de los métodos presentados: **Crop and Rescale** y **Compresión JPEG**. Ambas defensas son de las denominadas agnósticas al modelo por lo que tienen una implementación relativamente sencilla en la cual no se modifican los modelos subyacentes.

Si bien no son de las defensas más robustas existentes, nos serán de utilidad para analizar algunas de las características del *framework*, permitiéndonos evaluar el comportamiento de los distintos ataques ante estas defensas además del comportamiento de las defensas con imágenes limpias.

Tareas

Finalmente, para la evaluación de los algoritmos mencionados implementamos cuatro *Tasks* diferentes. Con estas tareas buscamos generar los datos necesarios para medir el desempeño de los ataques y defensas, además de permitirnos comparar nuestros propios resultados con los obtenidos por otros investigadores.

- **Train**: En primer lugar implementamos una *Task* para entrenar un modelo (potencialmente instancia de *Defense*) sobre un *DataSource* dado. Esta es una implementación bastante directa de descenso por gradiente estocástico con *momentum* y *cross entropy loss* como función de costo, que gracias a las funcionalidades proporcionadas por PyTorch resulta sumamente simple.

- **Accuracy**: Esta *Task* permite medir el desempeño de la red en tareas de clasificación, calculando el porcentaje de ejemplos clasificados correctamente además de la *top_k accuracy* para cierto k dado.

  Adicionalmente proporciona las normas L_0, L_2 y L_∞ promedio tanto de los ejemplos evaluados como de las perturbaciones adversarias generadas (en caso de usarse un ataque) y la confianza promedio de las predicciones realizadas entre otros valores. En el anexo 7.2 se detallan todos los resultados generados por esta tarea.

- **Samples**: Al trabajar con ejemplos adversarios no solo nos interesan los *fooling rates* obtenidos sino que también es deseable ver nosotros mismos qué tan perceptibles son las perturbaciones generadas. Esta *Task* permite visualizar el resultado de dichos ataques, almacenando tanto las imágenes limpias como los ejemplos adversarios y las propias perturbaciones, en conjunto a datos adicionales como las clases asignadas a cada ejemplo y las normas correspondientes.

- **Constrained Accuracy**: Por último incluimos otra tarea para medir el *accuracy* de las redes, pero en este caso para perturbaciones con normas acotadas. Esta *Task* implementa tanto el modo de ejecución simple como el múltiple, para los cuales recibe ya sea una lista de normas L_2 o de valores de *normalized dissimilarity* y procede iterativamente generando ejemplos adversarios que cumplan la restricción dada para cada caso, permitiéndonos visualizar el impacto de los ataques al utilizar perturbaciones de distintas magnitudes.

  La restricción de la norma fue incluida dentro de los propios algoritmos de ataque en los casos donde esto era posible y conveniente (BIM, MI-FGSM y DeepFool), mientras que para los restantes simplemente se obtiene el ejemplo adversario de forma normal y luego se escala su perturbación para llevarla a la norma correspondiente.

Los datos obtenidos son además graficados, ya sea en una única gráfica para el caso de ejecución simple o en potencialmente varias (una por cada defensa o modelo) para el caso de ejecución múltiple.

5.2. Resultados de ataques

Comenzamos por presentar una serie de resultados específicos obtenidos para los distintos ataques implementados. En estos experimentos buscamos evaluar algunos de los aspectos más relevantes de cada método, replicando las pruebas realizadas por sus autores y comparando los resultados a los obtenidos.

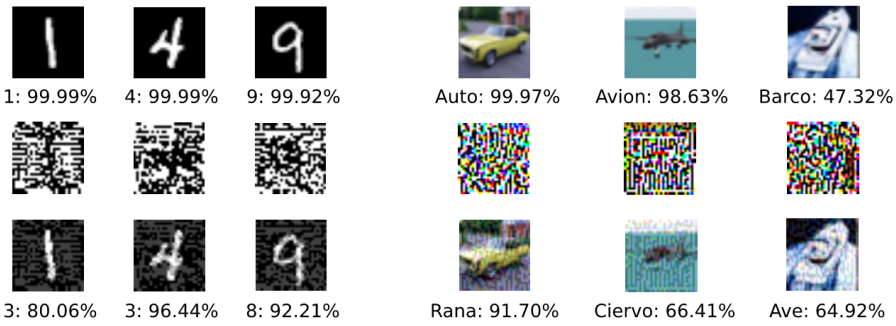


Figura 5.1: Ejemplos de perturbaciones generadas por el FGSM sobre los *datasets* MNIST y CIFAR-10 con valores de épsilon de 0,25 y 0,1 respectivamente (rango de 0 a 1). En cada caso se muestra la clase y probabilidad asignada por la red.

FGSM

Para evaluar este ataque nos limitamos a medir el *accuracy* de varias redes ante ejemplos adversarios generados con los mismos valores de ϵ reportados por los autores en el trabajo original, donde indican los índices de error obtenidos para los *datasets* MNIST y CIFAR-10 (aunque no en ImageNet, para el cual únicamente muestran un ejemplo generado).

En nuestras pruebas utilizamos las tres implementaciones de ResNet presentadas previamente, sobre las cuales obtuvimos índices de error muy cercanos a los reportados en el trabajo original (Cuadro 5.3) aunque con una confianza de predicción media más baja, lo cual puede deberse a que utilizamos arquitecturas diferentes. Esto de todas formas concuerda con las afirmaciones realizadas acerca de que es posible generar ejemplos adversarios efectivos y apenas perceptibles de forma sumamente eficiente.

Dataset	Epsilon	Error (%)	Confianza media (%)
MNIST	0,25	87,9 (89,4)	82,6 (97,6)
CIFAR-10	0,1	90,9 (87,15)	70,1 (96,6)
ImageNet	0,007	84.5	61.4

Cuadro 5.3: Índices de error y confianza de predicción obtenidos con el ataque FGSM sobre distintos *datasets*. Los valores de epsilon corresponden a imágenes en el rango $[0, 1]$. Entre paréntesis los valores del trabajo original.

BIM

Para este ataque recreamos una de las pruebas realizadas por sus autores en la cual comparan el desempeño de este método (y su variación *least-likely*) con la del FGSM. En dicha prueba miden el *accuracy* de una red sobre ejemplos adversarios generados para múltiples valores de épsilon, observando el impacto de cada método al aplicar perturbaciones de menor o mayor magnitud.

Los resultados obtenidos en nuestra evaluación se encuentran en la figura 5.2. Mientras que los valores obtenidos para el FGSM y el *least-likely* se asemejan a los del trabajo original, el *fooling-rate* obtenido por nuestra implementación del BIM es considerablemente más alto que el de los autores, disminuyendo rápidamente hasta cero el *accuracy* de la red, mientras que en el trabajo original este llega a un mínimo de aproximadamente 25 %.

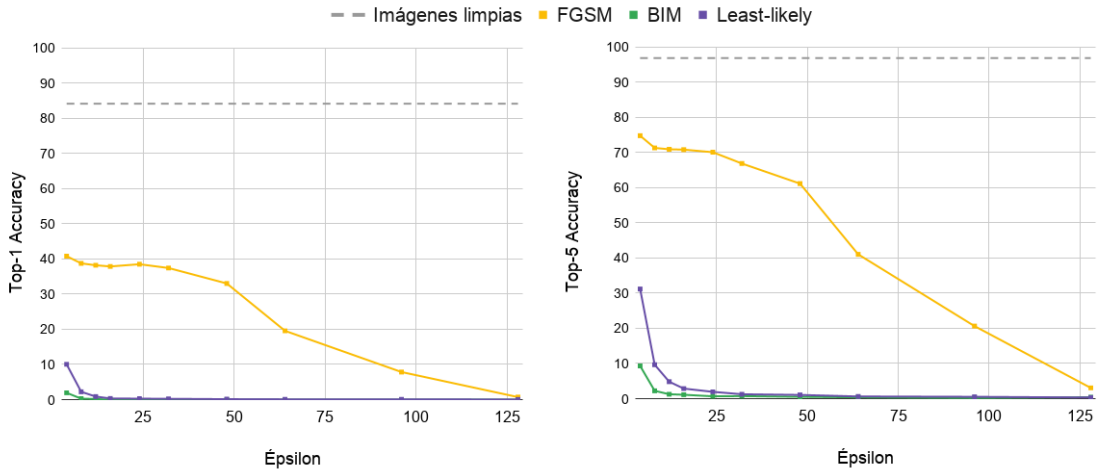


Figura 5.2: *Accuracy* obtenida por los métodos FGSM, BIM e *Iterative least-likely* en una red Inception-v3 para distintos valores de épsilon.

A pesar de haber realizado múltiples pruebas no logramos determinar el origen de esta discrepancia, pero dado a que en nuestros experimentos el BIM alcanza un *fooling-rate* incluso más alto que el reportado por los autores podemos concluir que se trata de un ataque sumamente efectivo en la modalidad *white-box*, capaz de generar perturbaciones verdaderamente imperceptibles.

MI-FGSM

Una de las principales características de este ataque es su capacidad para atacar modelos en modalidad *black-box*, por lo que parte de las pruebas realizadas por los autores se centra en evaluar la transferibilidad de los ejemplos generados sobre distintas redes. Para esto comparan este método con los dos anteriores, FGSM y BIM, usando el mismo valor de ϵ para todos los casos (equivalente a 16 en una escala de 1 a 256) y midiendo el *fooling-rate* obtenido sobre las varias arquitecturas de redes, todas entrenadas sobre ImageNet.

Para recrear este experimento hicimos uso de tres de las redes agregadas para ImageNet: ResNet-50, VGG-19 y Densenet-121, para las cuales obtuvimos los resultados presentados en el cuadro 5.4.

	Ataque	ResNet-50	VGG-19	Densenet-121
ResNet-50	FGSM	80.2	41.9	57.7
	BIM	**100.0**	44.8	73.8
	MI-FGSM	**100.0**	**60.0**	**82.3**
VGG-19	FGSM	55.7	80.6	57.2
	BIM	64.2	99.3	71.2
	MI-FGSM	**74.4**	**99.7**	**77.2**
Densenet-121	FGSM	53.3	43.7	84.7
	BIM	68.9	43.4	**100.0**
	MI-FGSM	**81.9**	**63.6**	99.8

Cuadro 5.4: *Fooling rates* (%) de los ataques FGSM, BIM y MI-FGSM sobre los modelos ResNet-50, VGG-19 y Densenet-121, tanto en modalidad *white-box* como *black-box* para un ϵ equivalente a 16 en el rango de 0 a 255.

Estos se asemejan a grandes rasgos a los obtenidos por los autores del ataque, donde el MI-FGSM alcanza un desempeño similar al del BIM en la modalidad *white-box* siendo superior para el caso *black-box*. De todas formas la ventaja de transferibilidad observada en nuestros experimentos no fue tan notable como la del trabajo original, dado que tanto el FGSM como el BIM presentan *fooling-rates* relativamente altos sobre los distintos modelos.

Estas diferencias pueden deberse al hecho de que los modelos usados en nuestros experimentos son diferentes a los del trabajo original, aunque quizás la mayor discrepancia a estudiar sea el el que en nuestras pruebas los ejemplos generados por el BIM se transfieren mejor que los del FGSM, contrario a los argumentos de los autores.

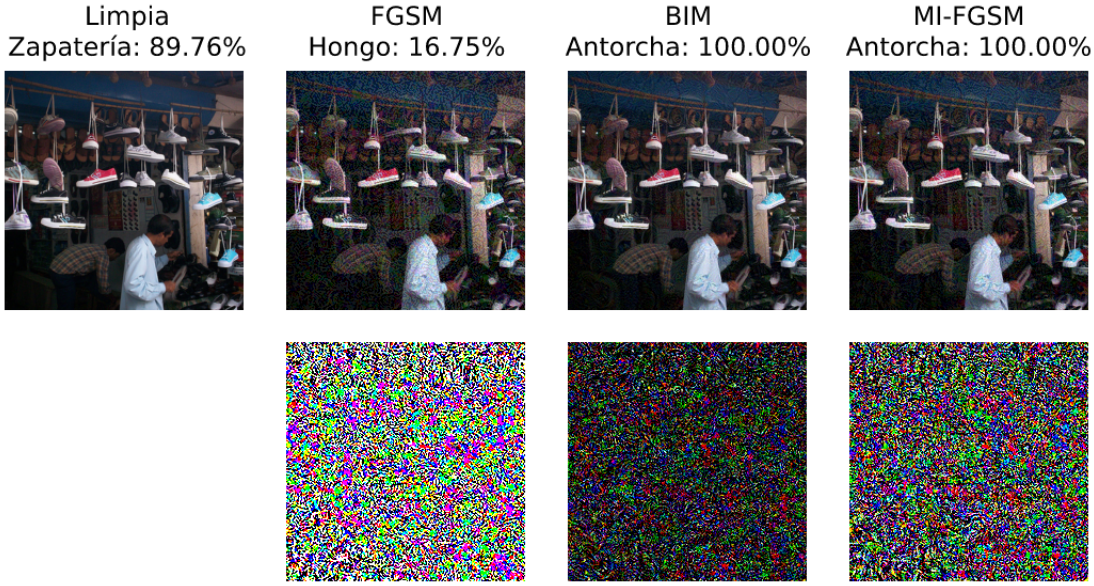


Figura 5.3: Ejemplos obtenidos por los ataques FGSM, BIM y MI-FGSM sobre una imagen de ImageNet, usando en todos los casos un épsilon de 16 en el rango de 1 a 256.

DeepFool

Este ataque es uno de los primeros propuestos y surge como respuesta a mejorar los algoritmos utilizados hasta ese momento, donde se tenía ataques muy buenos en encontrar

perturbaciones pequeñas a un costo alto computacional (3.4.1) o por lo contrario, ataques muy rápidos para generar ejemplos adversarios pero con perturbaciones sensiblemente más notorias (3.4.2).

Por esto realizan una comparación de las perturbaciones generadas por su algoritmo contra las obtenidas por FGSM y L-BFGS reportando lo que llaman *average robustness* o robustez media, y se corresponde al *normalized dissimilarity* para la norma L_2 (3.2). Reproducimos estas pruebas únicamente contra FGSM y comparamos con los valores del modelo que tiene *fooling rates* similares a los nuestros, obteniendo los resultados del cuadro 5.5.

		Deepfool	FGSM	ratio
MNIST	Original	$2{,}0 \times 10^{-1}$	1	0.200
	Nuestro	$0{,}98 \times 10^{-1}$	0,5	0.196
CIFAR10	Original	$3{,}0 \times 10^{-2}$	$1{,}3 \times 10^{-1}$	0.230
	Nuestro	$\mathbf{6{,}8 \times 10^{-3}}$	$0{,}5 \times 10^{-1}$	0.136
ImageNet	Original	$1{,}9 \times 10 - 3$	$4{,}7 \times 10 - 2$	0.040
	Nuestro	$1{,}3 \times 10 - 3$	$3{,}6 \times 10 - 2$	0.036

Cuadro 5.5: Comparación de los valores obtenidos de robustez adversaria en nuestra experimentación con los reportados en el informe original, mientras que para Deepfool es del ataque no dirigido.

Se puede apreciar que los valores de robustez obtenidos mantienen el mismo orden de magnitud que los reportados por los autores, siendo los nuestros incluso más bajos que los valores originales. Además, algo muy interesante es que también se conservaron las relaciones entre los valores de ambos ataques, como se muestra en la columna *ratio*.

Carlini & Wagner

Este es uno de los ataques más poderosos propuestos hasta el momento, logrando engañar a las redes mediante perturbaciones considerablemente pequeñas, sobre todo teniendo en cuenta que se trata de un ataque dirigido. Entre los experimentos realizados los autores comparan este método con algunos de los ataques más populares en ese entonces, demostrando que este logra generar perturbaciones adversarias hasta diez veces menor que las obtenidas con otros algoritmos.

Para corroborar los resultados reportados utilizamos nuestra implementación de este ataque en su versión para norma L_2, y siguiendo lo realizado en el informe original lo comparamos con Deepfool. En este caso la métrica usada es la norma L_2 media de las perturbaciones, teniendo en cuenta que ambos ataques logran engañar a la red utilizada en un 100 % de los casos.

Al ser este un ataque dirigido resulta relevante la elección de clase objetivo dado que impacta directamente sobre el tamaño de la perturbación obtenida, existiendo clases más difíciles de dirigir que otras. Teniendo en cuenta esto presentan sus resultados para tres elecciones de clase objetivo diferentes: mejor caso, caso promedio y peor caso. En nuestros experimentos utilizamos el caso promedio, que consiste en elegir la clase objetivo de forma aleatoria entre todas las clases incorrectas, obteniendo los resultados del cuadro 5.6.

Lo reportado para el caso promedio originalmente es bastante menor a las perturbaciones obtenidas con Deepfool, lo cual nosotros no logramos replicar. En nuestras pruebas ya obtenemos una media de perturbación para Deepfool bastante menor que el caso promedio para C&W original, pero este descenso no se manifiesta en nuestros resultados para

		C&W	Deepfool
MNIST	Original	1,76	2,11
	Nuestro	0,90	0,89
CIFAR10	Original	0,33	0,85
	Nuestro	0,25	0,19
ImageNet	Original	0,96	0,91
	Nuestro	0,38	0,22

Cuadro 5.6: Resultados de la media de la perturbación con L_2. La columna de C&W es el promedio del ataque dirigido utilizando clases aleatorias.

este ataque donde tenemos una media muy cercana a Deepfool e incluso hasta un poco mayor.

De todas maneras no se puede negar el poder que tiene este ataque sobre el resto. Aún sin haber obtenido una diferencia significativa contra Deepfool es necesario tener en cuenta que este se trata de un ataque dirigido, el cual es más difícil de obtener en comparación a ataques como Deepfool que son no dirigidos. Lo que también se demuestra con el tiempo de procesamiento que nos llevó cada prueba. Con los 1000 ejemplos generados utilizando Deepfool para CIFAR y MNIST se demoró menos de 5 minutos, mientras que los generados con C&W demoraron 30 y 70 minutos en obtenerse respectivamente [2]. Otro aspecto a tener en cuenta es que se realiza un postprocesamiento de redondeo para transformar las perturbaciones calculadas dentro del rango $[0, 1]$ al rango de imágenes válidas $[0, 255]$. Esto implica una pequeña degradación del ataque en lo que refería a la perturbación original.

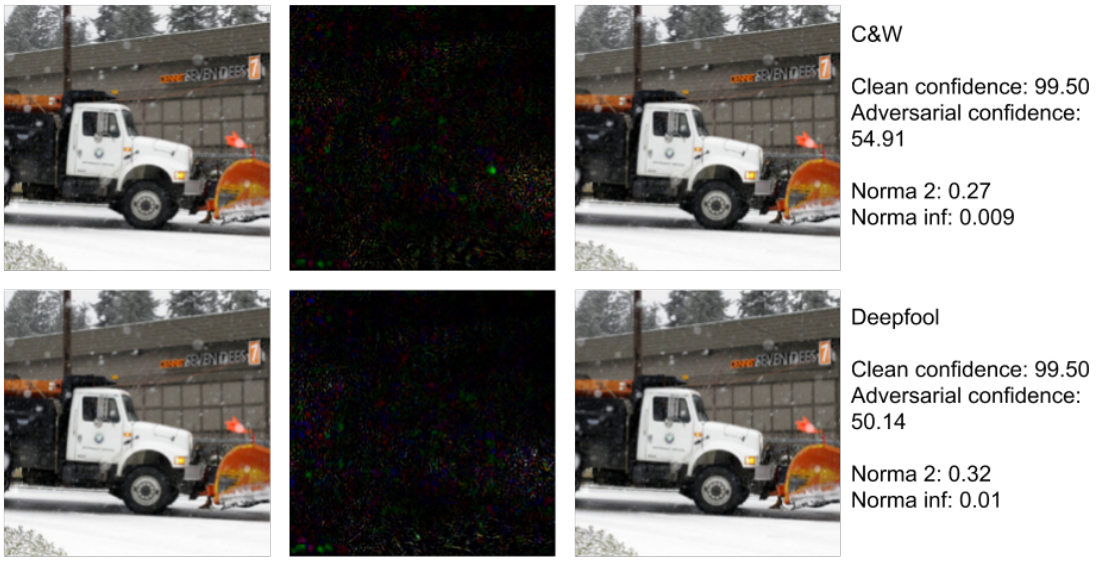


Figura 5.4: Ejemplos obtenidos por los ataques Deepfool y Carlini&Wagner sobre una imagen de ImageNet.

[2] Utilizando *abort_ early* para cortar la cantidad de iteraciones si no se esta convergiendo en la búsqueda de la mínima perturbación.

5.3. Comparación de Ataques

Para comparar los ataques implementados entre sí decidimos seguir el mismo enfoque usado para evaluar las defensas de pre-procesamiento presentadas en [32]. Este consiste en medir el *accuracy* de cada ataque forzándolos a generar perturbaciones que resulten en determinados valores de *normalized dissimilarity* (3.2).

Para esto utilizamos la red ResNet-50 entrenada sobre ImageNet y además agregamos un nuevo ataque *random noise* el cual agrega ruido aleatorio a los ejemplos de entrada, con el propósito de tomarlo como referencia al momento de comparar los otros algoritmos.

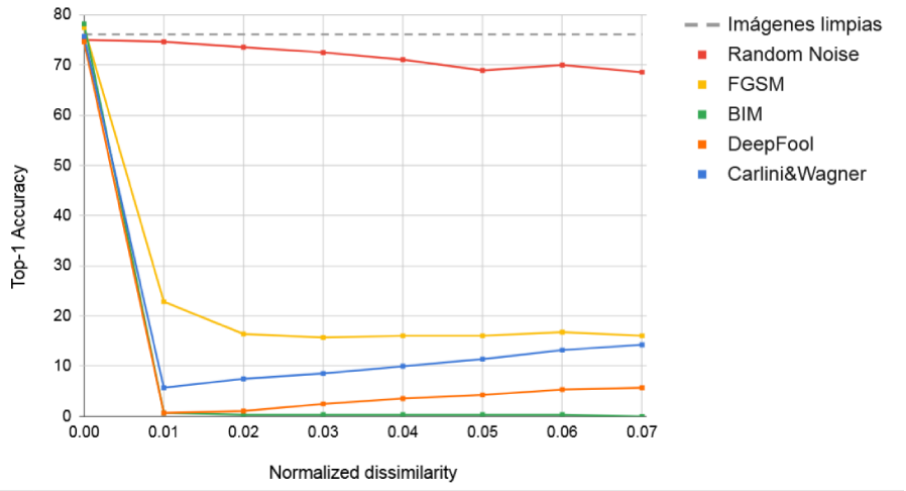


Figura 5.5: *Accuracy* de la red ResNet-50 sobre ejemplos adversarios de distintos ataques para múltiples valores de *normalized dissimilarity*.

Los resultados obtenidos se muestran en la figura 5.5. En esta se observa claramente la diferencia entre usar perturbaciones generadas por ataques adversarios y simplemente distorsionar imágenes de forma aleatoria, lo cual hace poco para disminuir el desempeño de la red.

También podemos observar la superioridad de los métodos iterativos sobre en este caso el FGSM (de un solo paso). Algo a tener en cuenta es que si bien el DeepFool y el BIM incluyen la restricción de norma dentro de sus algoritmos, para el Carlini&Wagner simplemente se escalan sus perturbaciones lo que puede explicar el desempeño levemente inferior que presenta.

5.4. Evaluación de Defensas

Para evaluar las defensas volvimos a usar la red ResNet-50, esta vez agregando el pre-procesamiento de los métodos *Crop and Rescale* y Compresión JPEG estudiados en [32]. Esto corresponde a una modalidad en la cual el atacante tiene conocimiento de la red utilizada pero no de los mecanismos de defensa aplicados, por lo que los ejemplos adversarios son generados utilizando únicamente la red atacada y su gradiente (sin el pre-procesamiento de la defensa).

Nuevamente obtuvimos resultados muy similares a los del trabajo original, en los cuales la Compresión JPEG logra mitigar parcialmente los efectos de perturbaciones pequeñas pero falla al proteger ante perturbaciones de mayor norma. La técnica *Crop and Rescale* por

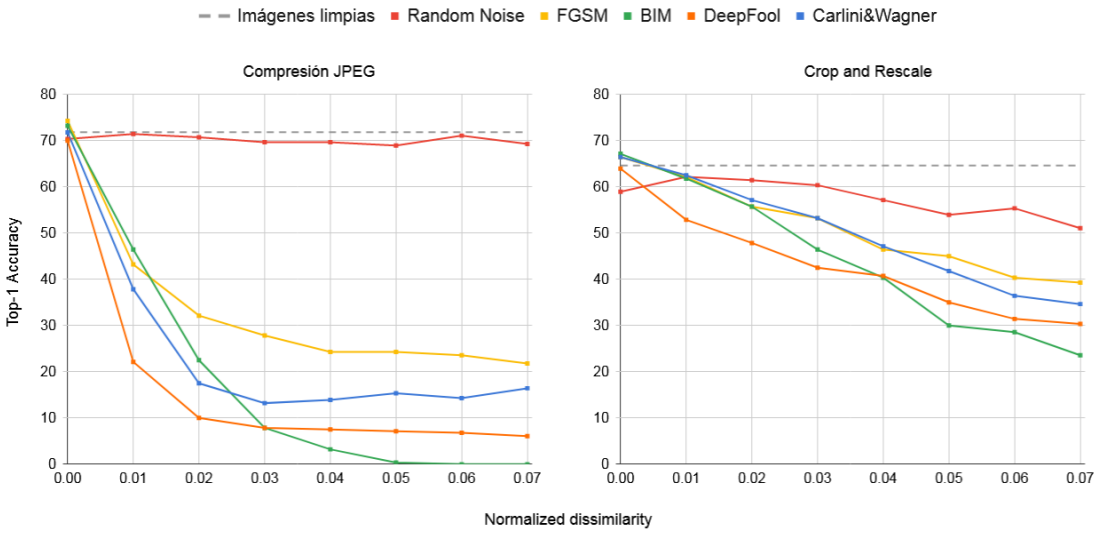


Figura 5.6: *Accuracy* obtenida para múltiples valores de *normalized dissimilarity* sobre imágenes generadas por los distintos ataques implementados en una red ResNet-50, utilizando los mecanismos de defensa **Compresión JPEG** (izquierda) y **Crop and Rescale** (derecha).

otro lado hace un mejor trabajo defendiendo ante ejemplos adversarios, aunque deteriora un poco más el desempeño sobre imágenes limpias.

Además volvimos a ejecutar este experimento con la misma configuración de ataques y defensas pero esta vez asumiendo que los atacantes tienen conocimiento de los mecanismos de defensa utilizados. De esta forma se utilizan los ataques de forma análoga a lo propuesto en la sección 3.4.8. Para el caso de Compresión JPEG se aplica BPDA (caso especial) aproximando el gradiente con la función identidad, mientras que para *Crop and Rescale* aplicamos el ataque EOT de la sección 3.4.7, de forma de considerar múltiples recortes al momento de generar las perturbaciones.

La figura 5.7 contiene los resultados del experimento. En este caso el *accuracy* obtenido para las distintas normas se asemeja mucho más a los obtenidos con la red sin defensas, corroborando las afirmaciones acerca de qué mecanismos basados en la ofuscación de gradientes pueden ser burlados mediante las técnicas mencionadas [26].

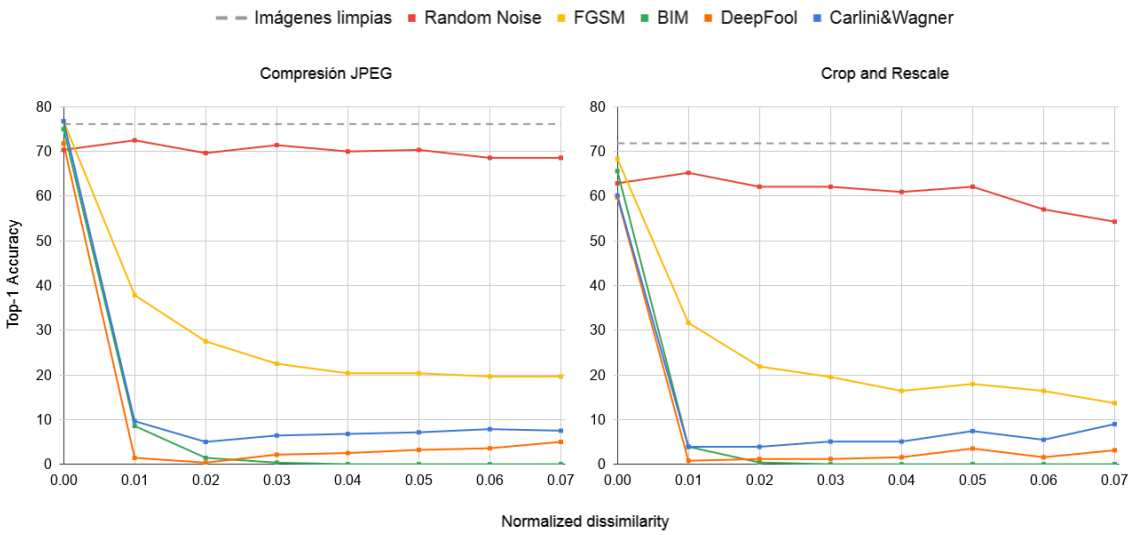


Figura 5.7: *Accuracy* obtenida para múltiples valores de *normalized dissimilarity* sobre imágenes generadas por los distintos ataques implementados en una red ResNet-50, utilizando los mecanismos de defensa **Compresión JPEG** (izquierda) y **Crop and Rescale** (derecha) en modalidad *white-box*, aplicando BPDA y EOT respectivamente.

Capítulo 6

Conclusiones

En este proyecto nos propusimos abordar un problema relativamente reciente que afecta a una de las áreas más relevantes de la computación en la actualidad como lo son las redes neuronales y el aprendizaje automático. Para esto nos centramos en el estudio de los ejemplos adversarios en el contexto de visión artificial, más concretamente sobre el problema de clasificación de imágenes. El hecho de tratarse de un problema relativamente simple (aunque sirve como pilar para otros problemas más complejos) nos permitió centrarnos en el fenómeno estudiado propiamente más que en la tarea subyacente.

Debido a la gran cantidad de usos que se les da hoy en día a las redes neuronales resulta de sumo interés el comprender las limitaciones y vulnerabilidades que presentan con el fin de poder mitigarlas y obtener las garantías que los sistemas críticos de la actualidad necesitan. En este aspecto, el estudio de los ejemplos adversarios no solo nos ayuda a estudiar la robustez de estos modelos sino que también permite obtener un mayor entendimiento de las redes profundas en general.

El que se tratase de un tema reciente y en activa investigación dificultó el análisis del estado del arte, sobre todo considerando la frecuencia con la cual se publican trabajos nuevos. A esto se suma el hecho de que muchas veces en estos artículos se llegaban a conclusiones contradictorias o que invalidaban afirmaciones de trabajos anteriores. De todas formas creemos que el trabajo realizado brinda un panorama relativamente actualizado y abarcativo acerca de los ejemplos adversarios y los riesgos que representan para los sistemas de la actualidad.

A pesar de los distintos mecanismos de defensa propuestos aún no se ha logrado garantizar la robustez de las redes ante este tipo de ataques. Si bien existen defensas que logran mejorar el desempeño sobre ejemplos adversarios, solo lo hacen de forma parcial y en ocasiones al costo de un deterioro en el desempeño sobre ejemplos limpios. A esto se suma el hecho de que para argumentar robustez las defensas deben proteger no sólo contra ataques existentes sino que también contra ataques futuros. Esto ha probado ser una tarea difícil dada la cantidad de defensas propuestas hasta el momento que han sido posteriormente vulneradas por atacantes, los cuales con conocimiento de los mecanismos aplicados logran quebrantarlas. Por otro lado, la capacidad de transferencia de los ejemplos adversarios entre distintas redes permite a los atacantes engañar sistemas de los cuales se posee poco o nulo conocimiento. Esto combinado al hecho de que es posible llevar dichos ejemplos al mundo físico hacen de este fenómeno un problema de interés no solo en lo académico sino también dentro de la industria, suponiendo un riesgo serio que puede afectar a los sistemas del mundo real.

Mediante los experimentos realizados logramos en general corroborar las afirmaciones da-

das por los respectivos autores. Además de poder replicar muchos resultados, el hecho de haber implementado nuestras propias versiones de los algoritmos nos permitió adquirir un mayor entendimiento del problema, como también sobre las redes neuronales en general. Una de las mayores dificultades al momento de evaluar y comparar los distintos algoritmos fue lidiar con la gran cantidad de factores involucrados, tanto en la configuración de los modelos y los métodos como en los mecanismos de evaluación. En este contexto las pruebas realizadas contienen algunas diferencias con respecto a las pruebas presentadas por los autores, las cuales nos permitieron simplificar algunos aspectos de implementación y obtener una mayor flexibilidad en su uso. De todas formas creemos que estas diferencias no son lo suficientemente relevantes como para generar discrepancias significativas en las conclusiones obtenidas.

Durante nuestra evaluación de los ataques y defensas el *framework* implementado tuvo un rol muy importante. Nos permitió realizar una gran cantidad de pruebas sobre distintas combinaciones de algoritmos y configuraciones de parámetros de forma relativamente rápida y sencilla. Creemos que muchas de las consideraciones tomadas resultaron de suma utilidad al momento de llevar a cabo varias de las tareas necesarias (no solo las relacionadas a ejemplos adversarios), como por ejemplo el almacenamiento estructurado y con información contextual de los resultados que efectivamente nos simplificó su posterior análisis, o el uso de los archivos de tareas que permiten almacenar y reproducir fácilmente experimentos de interés. Además entendemos que la estructura modular del *framework* lo hace particularmente flexible y escalable, permitiéndonos incorporar nuevos componentes (como redes de arquitecturas diferentes y ataques nuevos) en etapas finales del proyecto y haber sido capaces de evaluarlos rápidamente en conjunto a otros componentes ya existentes.

Desde el comienzo de la implementación del *framework* tuvimos como objetivo que este también fuera de utilidad para otros investigadores en el área. Por este motivo todo el código desarrollado, incluyendo los resultados obtenidos, se encuentran disponibles en github[1] donde esperamos pueda ser mejorado con la ayuda de otros usuarios de la comunidad.

6.1. Trabajo Futuro

Como trabajo a futuro planteamos la posibilidad de utilizar los conocimientos recabados y la herramienta desarrollada para continuar con el desarrollo de nuevos mecanismos de ataque y de defensa. Especialmente utilizando enfoques diferentes a los propuestos hasta el momento. Entre cosas a evaluar sugerimos el uso de nuevas métricas de distancia que tengan más en consideración la percepción humana, además de las ya usadas normas L_p. Como una opción proponemos el aplicar técnicas de transferencia de estilos durante la optimización, con el fin de obtener imágenes que aparenten ser ejemplos naturales para un observador humano pero sean interpretados como clases completamente diferentes por una red.

Otra línea de trabajo paralela consiste en mejorar el *framework* implementado de forma que este sea una herramienta completa que resulte de utilidad a otros investigadores. Detro de las posibles mejoras se incluye la implementación de múltiples ataques y defensas existentes que permitan realizar pruebas más abarcativas. También la incorporación de funcionalidades que simplifiquen el estudio de los ejemplos adversarios en otras áreas distinta al procesamiento de imágenes, como pueden ser el procesamiento del lenguaje natural y reconocimiento de voz.

[1] https://github.com/diegoirigaray/CrAdv

Bibliografía

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[2] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[3] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *CoRR*, abs/1603.06042, 2016.

[4] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv e-prints*, page arXiv:1312.6199, Dec 2013.

[5] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv e-prints*, page arXiv:1412.6572, Dec 2014.

[6] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv e-prints*, page arXiv:1607.02533, Jul 2016.

[7] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on machine learning models. *CoRR*, abs/1707.08945, 2017.

[8] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1528–1540, New York, NY, USA, 2016. ACM.

[9] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. *arXiv e-prints*, page arXiv:1610.08401, Oct 2016.

[10] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[11] Intuitively understanding convolutions for deep learning. `https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1`.

[12] Ravindra Parmar. Detection and segmentation through convnets. `https://towardsdatascience.com/detection-and-segmentation-through-convnets-47aa42de27ea`, Sep 2018.

[13] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. ZOO: Zeroth Order Optimization based Black-box Attacks to Deep Neural Networks without Training Substitute Models. *arXiv e-prints*, page arXiv:1708.03999, Aug 2017.

[14] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models. *arXiv e-prints*, page arXiv:1712.04248, Dec 2017.

[15] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. *arXiv e-prints*, page arXiv:1608.04644, Aug 2016.

[16] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv e-prints*, page arXiv:1605.07277, May 2016.

[17] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting Adversarial Attacks with Momentum. *arXiv e-prints*, page arXiv:1710.06081, Oct 2017.

[18] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. On the intriguing connections of regularization, input gradients and transferability of evasion and poisoning attacks. *CoRR*, abs/1809.02861, 2018.

[19] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer, New York, NY, USA, second edition, 2006.

[20] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble Adversarial Training: Attacks and Defenses. *arXiv e-prints*, page arXiv:1705.07204, May 2017.

[21] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings. *arXiv e-prints*, page arXiv:1511.07528, Nov 2015.

[22] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: a simple and accurate method to fool deep neural networks. *arXiv e-prints*, page arXiv:1511.04599, Nov 2015.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec 2014.

[24] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. *CoRR*, abs/1707.07397, 2017.

[25] Anish Athalye. Fooling neural networks in the physical world with 3d adversarial objects. `https://www.labsix.org/physical-objects-that-fool-neural-nets/`, Oct 2017.

[26] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. *arXiv e-prints*, page arXiv:1802.00420, Feb 2018.

[27] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial Patch. *arXiv e-prints*, page arXiv:1712.09665, Dec 2017.

[28] Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi. One pixel attack for fooling deep neural networks. *arXiv e-prints*, page arXiv:1710.08864, Oct 2017.

[29] Sara Sabour, Yanshuai Cao, Fartash Faghri, and David J. Fleet. Adversarial Manipulation of Deep Representations. *arXiv e-prints*, page arXiv:1511.05122, Nov 2015.

[30] Andras Rozsa, Ethan M. Rudd, and Terrance E. Boult. Adversarial diversity and hard positive generation. *CoRR*, abs/1605.01775, 2016.

[31] Yang Song, Taesup Kim, Sebastian Nowozin, Stefano Ermon, and Nate Kushman. PixelDefend: Leveraging Generative Models to Understand and Defend against Adversarial Examples. *arXiv e-prints*, page arXiv:1710.10766, Oct 2017.

[32] Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens van der Maaten. Countering Adversarial Images using Input Transformations. *arXiv e-prints*, page arXiv:1711.00117, Oct 2017.

[33] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial networks. *ArXiv*, abs/1406.2661, 2014.

[34] Zico Kolter and Aleksander Madry. Adversarial robustness - theory and practice. `https://adversarial-ml-tutorial.org/`, Dec 2018.

[35] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. *arXiv e-prints*, page arXiv:1706.06083, Jun 2017.

[36] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks. *arXiv e-prints*, page arXiv:1511.04508, Nov 2015.

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[38] Justin Gilmer, Luke Metz, Fartash Faghri, Samuel S. Schoenholz, Maithra Raghu, Martin Wattenberg, and Ian J. Goodfellow. Adversarial spheres. *CoRR*, abs/1801.02774, 2018.

[39] Why data normalization is necessary for machine learning models. `https://medium.com/@urvashilluniya/why-data-normalization-is-necessary-for-machine-learning-models-681b65a05029`.

[40] Torchvision models. `https://pytorch.org/docs/stable/torchvision/models.html`.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[42] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[43] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

Capítulo 7

Anexos

En este anexo se agregan detalles de relevancia sobre la implementación del *framework* desarrollado. Toda la documentación restante como el manual de uso se encuentra en el repositorio de código público.

7.1. Archivo de tareas

En este anexo presentamos el esquema completo de los archivos de tareas utilizados para la especificación de experimentos en el *framework*. Este esquema indica la estructura esperada para los archivos además de una lista de todos los argumentos posibles con sus respectivos tipos y descripciones.

Teniendo en cuenta que se utiliza el formato *json*, cada archivo se compone de un objeto con la estructura presentada a continuación.

- **config** (Object): Configuración para la ejecución de las tareas.
  - **device** (String): Identificador del *torch.device* a utilizar durante la ejecución. Puede ser 'cpu' o 'gpu', seguido opcionalmente por ':X' donde X es el número identificador del dispositivo correspondiente.
  - **device_ids** (Array): Lista de *torch.device* pasada al *DataParallel* de PyTorch cuando el parámetro **multi_gpu** es verdadero. Permite especificar las gpu's a utilizar.

    [Items] (String): Identificador del dispositivo.
  - **multi_gpu** (Boolean): *Flag* para activar/desactivar el uso de múltiples gpu's. Por defecto usa todas las gpu's disponibles.
  - **results_path** (String): Ruta en la cual guardar los resultados generados durante la ejecución. Por defecto los guarda dentro de la carpeta *results* del repositorio.
  - **safe_mode** (Boolean): *Flag* para activar/desactivar la propagación de errores. Por defecto los errores generados durante la ejecución de cualquier tarea detienen la ejecución de todo el algoritmo. Poner esta *flag* en *true* hace que el *Scheduler* capture y guarde cualquier error generado permitiendo continuar con la ejecución del resto de las tareas.

- **tasks*** (Array): Lista de tareas a ejecutar.

  [Items] (Object): Especificación para la ejecución de una tarea. Incluye la información de la propia *Task* y la de los modelos, *DataSources*, *Attacks* y *Defenses* con los cuales será ejecutada.

  - **task_data*** (Object): Especificación de la *Task* a usar.
    - **task_name** (String): Nombre identificador de la *Task* a ejecutar.
    - **task_params** (Object): Diccionario pasado como *kwargs* al inicializar la *Task*.
    - **attack_on_defense** (Boolean): *Flag* para elegir qué componente se pasa a los ataques. Cuando se evalúan ataques y defensas de forma simultánea, los ataques pueden recibir tanto la instancia del modelo original sin defender como la instancia de la defensa. Por defecto ('true') se pasa la instancia de la defensa, cambiar el valor de esta *flag* a 'false' para utilizar la instancia del modelo original, útil cuando la defensa no es diferenciable o si se quiere evaluar el ataque sin conocimiento de la defensa.
    - **exec_multi** (Boolean): *Flag* para elegir entre modos de ejecución. Por defecto se utiliza el modo de ejecución simple de las tareas. Cambiar esta *flag* a 'true' para utilizar el modo de ejecución múltiple.
    - **plot_keys** (Array): Permite especificar resultados a graficar automáticamente. Al utilizar el modo de evaluación de ataques en las tareas (y asumiendo que la tarea retorna un diccionario), se puede especificar en esta lista las claves a graficar del resultado. El contenido de los elementos especificados debe ser numérico.

      [Items] (String): Clave de los resultados a graficar.
    - **plot_together** (Boolean): *Flag* que permite indicar si los resultados especificados en **plot_keys** deben ser graficados en la misma gráfica.
    - **skip_no_attack** (Boolean): *Flag* que permite activar/desactivar la ejecución de tareas sin ataques. Por defecto, las tareas son ejecutadas primero sin ataque y luego para cada ataque especificado. Cambiar esta *flag* a 'true' para evitar la ejecución sin ataque.
    - **skip_no_attack_variables** (Boolean): *Flag* que permite activar/desactivar la ejecución de tareas sin variables de ataques. Por defecto, las tareas son ejecutadas primero con sus parámetros iniciales y luego con variando cada uno de los hiper-parámetros especificados. Cambiar esta *flag* a 'true' para evitar la ejecución utilizando parámetros iniciales.
    - **skip_no_defense** (Boolean): *Flag* que permite activar/desactivar la ejecución de tareas sin defensas. Por defecto, las tareas son ejecutadas primero sin defensa y luego para cada defensa especificada. Cambiar esta *flag* a 'true' para evitar la ejecución sin defensa.
  - **nets*** (Array): Modelos y *DataSources* a utilizar en la ejecución de la tarea.

    [Items] (Object): Especificación del modelo y su respectivo *DataSource*.
    - **model_name** (String): Nombre identificador del modelo a utilizar.

- **model_params** (Object): Diccionario pasado como *kwargs* al inicializar el modelo.
- **datasource_name** (String): Nombre identificador del *DataSource* a utilizar.
- **datasource_params** (Object): Diccionario pasado como *kwargs* al inicializar el *DataSource*.
- **net_id** (String): Identificador de los pesos a utilizar en el modelo. El *Parser* buscará un archivo llamado '[net_id].pth' dentro del directorio *data/weights/* del repositorio y en caso de encontrarlo cargará sus pesos al modelo.

- **defenses*** (Array): Defensas a utilizar en la ejecución de la tarea.

  [Items] (Object): Especificación de un componente *Defense*.
  - **defense_name** (String): Nombre identificador de la *Defense* a utilizar.
  - **defense_params** (Object): Diccionario pasado como *kwargs* al inicializar la *Defense*.

- **attacks*** (Array): Ataques a utilizar en la ejecución de la tarea.

  [Items] (Object): Especificación de un componente *Attack*.
  - **attack_name** (String): Nombre identificador del *Attack* a utilizar.
  - **attack_params** (Object): Diccionario pasado como *kwargs* al inicializar el *Attack*.
  - **except_variables** (Array): Variables de ataque a ignorar con este ataque. Si se especificaron variables de ataque mediante el parámetro *attack_variables* de las cuales algunas no aplican a este ataque, agregarlas a esta lista.

    [Items] (String): Nombre de las variables a ignorar, correspondientes a su *variable_name*.
  - **specific_models** (Array): Lista de modelos sobre los cuales ejecutar el ataque, simulando un ataque en modalidad *black-box* con modelo sustituto.

    [Items] (Object): Misma estructura a la descrita para el parámetro **nets**.
  - **on_task_model** (Boolean): Por defecto, los ataques se ejecutan primero sobre el modelo/defensa correspondiente a la tarea y luego para todos los modelos especificados en *specific_models*. Cambiar esta *flag* a 'false' para evitar la ejecución sobre el modelo de la tarea.

- **attack_variables** (Array): Lista de hiper-parámetros a usar con el modo de evaluación de ataques de la tarea.

  [Items] (Object): Nombre y valores de cada hiper-parámetro.
  - **variable_name** (String): Nombre del hiper-parámetro, correspondiente a algún atributo del ataque.
  - **variable_values** (Array): Lista de valore a evaluar.

    [Items] (Any): Valor a evaluar en el hiper-parámetro.

7.2. Tarea *accuracy*

En este anexo se listan y detallan todos los parámetros resultantes de la *task accuracy* desarrollada.

- *total*: Total de imágenes procesadas en la ejecución.*top_k accuracy*
- *correct*: La cantidad de imágenes predecidas correctamente por la red, de no existir ataque es sobre el conjunto original, sino sobre las imágenes perturbadas resultantes del ataque.
- *adversarial*: La cantidad de imágenes bien clasificadas correctamente, que luego del ataque cambiaron su predicción.
- *c_total*: El total de las imágenes originales bien clasificadas.
- *c_accuracy*: El porcentaje de las imágenes correctamente clasificadas luego del ataque, que originalmente fueron correctamente predecidas.
- *correct_avg_confidence*: El promedio de la confianza en la predicción de la clase correcta, de ser esta la salida de la red.
- *fooled_avg_confidence*: El promedio de la confianza en la predicción dada para los ejemplos adversarios.
- *dataset_norm_0*: algo
- *dataset_avg_norm_2*: promedio de L_2 para el conjunto de imágenes original.
- *dataset_avg_norm_inf*: promedio de L_∞ para el conjunto de imágenes original.
- *adv_avg_norm_0*: promedio de L_0 para el conjunto de imágenes con perturbación adversaria.
- *adv_avg_norm_2*: promedio de L_2 para el conjunto de imágenes con perturbación adversaria.
- *adv_avg_norm_inf*: promedio de L_∞ para el conjunto de imágenes con perturbación adversaria.
- *adv_disimilarity*: cálculo de 3.2 para L_2
- *adv_inf_disimilarity*: cálculo de 3.2 para L_∞