"AI" demystified: ~~a decompiler~~

Traducción de Aitor Saiz Lasheras con el permiso, pero sin la revisión, de Giacomo Tesio. Los errores de traducción son sólo míos, no del autor del artículo. Original aquí: https://encrypted.tesio.it/2021/09/01/a_decompiler_for_artificial_neural_networks.html

Ver también la entrada ¿Qué es la informática? Por un conocimiento humano sin cajas negras.

ERRATA CORRIGE

3 de septiembre de 2021

Resulta que las variaciones de los *pesos* (gradientes), registradas **aquí** durante la fase de compilación, son suficientes para reconstruir el conjunto de datos original.

En concreto, el gradiente registrado desde el nodo de la primera capa es, en sí mismo, una transformación lineal de la entrada.

Esto significa que, tal y como está ahora, **decompile.py** no prueba ni refuta ninguna afirmación sobre el "modelo" de una "red neuronal artificial".

Sigo respaldando el resto del artículo y sigo creyendo que cualquier "red neuronal artificial" es simplemente una máquina de mapeo vectorial virtual programada estadísticamente que **canaliza** filtros compuestos por reductores vectoriales.

Además, sigo creyendo que el software ejecutado por tales máquinas (el "modelo" en la jerga de IA/ML) es una obra derivada del conjunto de datos de origen, al igual que el binario compilado a partir de un código fuente en C es una obra derivada del código fuente.

Aunque este descompilador no demuestra que una parte sustancial del código fuente esté contenida en el binario de salida, sigo creyendo que leer, ejecutar y depurar el compilador proporciona una visión profunda de dicha derivación porque:

produce un binario ejecutable para una máquina de destino específica a partir de un conjunto de datos de origen y, por lo tanto, no es solo una forma de *hash* o compresión con pérdida
no es una forma de cifrado porque no se utiliza una clave de cifrado

Las prácticas del campo pueden fusionar en un único proceso la ingeniería de estas máquinas virtuales y su programación simplemente porque, al fin y al cabo, tanto las máquinas virtuales como el "modelo" que ejecutan... son piezas de software.

> Además, la "primavera de la IA" generó un gran revuelo y varios incentivos políticos para preservar el confuso lenguaje antropomórfico en lugar de uno más sencillo, más fácil de entender para todo el mundo.
>
> Dicho esto, lamento mucho las afirmaciones erróneas sobre la descompilación.

Resumen

Para demostrar que cualquier "red neuronal artificial" es simplemente una máquina (virtual) programada estadísticamente cuyo software "*modelo*" (aquí el autor tacha la palabra en el original, para denotar la inconveniencia del uso de este término, N.T.) es una obra derivada del conjunto de datos de origen utilizado durante su "entrenamiento", proporcionamos un pequeño conjunto de herramientas para ensamblar y programar dichas máquinas, así como un descompilador que reconstruye el conjunto de datos de origen a partir de las matrices crípticas que constituyen el software ejecutado por ellas.

Por último, probamos el conjunto de herramientas en el clásico conjunto de datos MNIST y comparamos el conjunto de datos descompilado con el original.

Palabras clave

Inteligencia Artificial, Aprendizaje Automático, Redes Neuronales Artificiales, Microsoft, GitHub Copilot, Python, Programación Estadística, Máquina de Mapeo Vectorial

Introducción

A pesar del lenguaje antropomórfico confuso y la narrativa poco profesional, en los últimos años el campo de la Informática que se conoce con los nombres de "Inteligencia Artificial" y "aprendizaje automático" ha acumulado un largo historial de éxitos autoproclamados. Detrás del bombo publicitario y las representaciones de ciencia ficción de futuros maravillosos en los que humanos virtuales explorarán el Universo en nombre de una humanidad casi extinta, difundidas para frenar cualquier solución efectiva que pudiera salvar las vidas de miles de millones de personas reales, hay unas pocas técnicas mal entendidas (y a menudo mal utilizadas) que realmente aporten algo de innovación.

La más emblemática de estas innovaciones es la denominada, de forma impropia, "red neuronal artificial", en referencia a un modelo erróneo del cerebro humano que inspiró su creación allá por los años sesenta. A partir de esta clase de herramientas, compuestas y programadas de múltiples maneras, las grandes corporaciones con acceso a enormes conjuntos de datos, dinero y potencia computacional están comercializando varios servicios *útiles*.

Entre ellos, Microsoft ha lanzado recientemente GitHub Copilot, un Servicio como Sustituto del Software que sugiere a los programadores apresurados fragmentos de código extraídos y adaptados automáticamente de la vasta base de código subida a GitHub. Desde el principio, Microsoft reconoció que Copilot puede *recitar* código literalmente de los repositorios de GitHub en el 0,1 % de los casos, pero no tardó mucho en que a un desarrollador se le sugiriera una pieza de software

libre especialmente valiosa con una licencia permisiva en lugar del *copyleft* estricto decidido por los autores.

Tras la evidente reacción negativa de la comunidad del software libre, varios grupos de presión, abogados e incluso algunos políticos se apresuraron a justificar a la pobre empresa, argumentando básicamente que utilizar obras protegidas por derechos de autor para el "aprendizaje automático" es en realidad justo, por lo que los desarrolladores no deberían molestar a la corporación.

Como siempre, el problema es que las personas que no entienden realmente cómo funciona el "aprendizaje automático" confiarán y difundirán las palabras de esos "expertos" mal informados, y esto perjudicará inevitablemente a los explotados "humanos en el bucle".

De hecho, Microsoft no utilizó la enorme cantidad de código fuente de alta calidad de sus propios productos para entrenar a Copilot, sino el software libre y el software *open source* subido a GitHub. Esto a pesar de que solo una minúscula minoría de proyectos en GitHub pasa por las largas revisiones y los controles de calidad que una empresa como Microsoft puede permitirse establecer, y desde Heartbleed todos sabemos que la "ley de Linus" no es más que un cuento para niños ingenuos.

No hay "aprendizaje" en una "red neuronal artificial"

La mayoría de los argumentos a favor del nuevo servicio de Microsoft provienen de personas que realmente no entienden el funcionamiento interno de una "red neuronal artificial". Una RNA es simplemente una máquina virtual diseñada para ser programada estadísticamente, mediante ejemplos.

Este tipo de máquinas virtuales se componen de diminutos dispositivos que se denominan, de forma impropia, "neuronas artificiales" o "perceptrones", pero que son simplemente reductores de vectores que, dada una función diferenciable no lineal, pliegan un vector de un determinado espacio N-dimensional en uno unidimensional.

Un vector, como sabrás, no es más que una lista de números que un ser humano puede interpretar como una descripción de algo. Por ejemplo, a mí se me podría describir mediante (40, 20, 16), que representan, respectivamente, mi edad, el número de años que he trabajado como programador y el número de lenguajes de programación que conozco.

El funcionamiento interno de un reductor vectorial es trivial: basta con calcular el producto escalar entre el vector de entrada y uno paramétrico y, a continuación, aplicar la función diferenciable para obtener la magnitud del vector de salida; pero lo interesante es cómo se puede programar estadísticamente el vector paramétrico distribuyendo los errores observados en un conjunto de datos conocido, de modo que, con un poco de suerte, tras algunas iteraciones sobre el conjunto de datos, el reductor vectorial podría aproximarse a la salida deseada para un vector dado.

Estos reductores vectoriales pueden combinarse fácilmente en diversas topologías y programarse de múltiples formas, de modo que toda una "red neuronal artificial" puede programarse estadísticamente ("entrenarse", en la jerga de la IA/ML) para aproximarse a una de las posibles transformaciones de un espacio vectorial a otro.

Sin embargo, no hay ningún "aprendizaje" en curso: solo el ajuste iterativo de vectores paramétricos para aproximarse a una salida determinada. Dado que las diversas entradas de los vectores paramétricos suelen denominarse *"pesos"*, las personas que intentan evitar la jerga antropomórfica de la IA suelen llamar al proceso *"calibración"*, pero, como veremos, es importante utilizar en su lugar el término "programación estadística" para aclarar a las personas sin formación los componentes, los procesos, las funciones y las responsabilidades que caracterizan a dicha tecnología.

De hecho, la única "inteligencia artificial" en una máquina de mapeo vectorial es aquella que se nos ha enseñado a ver (de manera imaginaria) mediante un marketing agresivo, una propaganda confusa y un lenguaje técnico deficiente.

Máquinas de mapeo vectorial: ingeniería y programación

Las máquinas de mapeo vectorial (conocidas erróneamente como "redes neuronales artificiales") se han implementado habitualmente como software, por lo que son básicamente máquinas virtuales especializadas (y opacas).

En un intento por preservar la quimera de la "inteligencia artificial", el diseño de estas máquinas se ha confundido tanto con su programación, que hemos tenido que inventar definiciones hiperbólicas como "hiperparámetros" para permitir que los "científicos de datos" distingan y se comuniquen sobre el diseño de una máquina específica, su programación (denominada erróneamente "entrenamiento") y su funcionamiento interno en tiempo de ejecución.

Sin embargo, si nos deshacemos de la narrativa de ciencia ficción y hablamos de una máquina virtual opaca especializada en mapeos aproximados entre espacios vectoriales, podemos basarnos en el lenguaje habitual de la informática para describir y comprender estas herramientas y su impacto.

De hecho, diseñar una máquina de mapeo vectorial es un acto de ingeniería compuesto por varios pasos, como decidir el número y los tipos de filtros, sus conexiones, la función diferenciable que se va a utilizar, el proceso de inicialización de los vectores paramétricos de cada reductor vectorial, etc.

El resultado de dicho proceso de ingeniería es una máquina que puede construirse en hardware o emularse mediante software, pero que resulta tan útil como un ordenador sin ningún software instalado.

Es importante señalar que, cuando dicha máquina virtual se emula mediante software, la programación de dicho emulador otorga obviamente a los autores un derecho de autor sobre el mismo, al igual que ocurre con la programación de cualquier otra máquina virtual. Pero, al igual que con cualquier otra máquina virtual, el titular de los derechos de autor del software que emula la máquina no puede reclamar derechos sobre el software ejecutado por dicha máquina.

De hecho, a pesar de la confusión generada por la jerga y las prácticas primitivas, la ingeniería de las máquinas de mapeo vectorial es un proceso totalmente distinto y diferente al de su programación.

La máquina de mapeo vectorial es un artefacto intelectual diferente del programa que ejecuta y constituye una obra creativa por sí misma. De hecho, la misma máquina puede *programarse* para producir diferentes tareas de mapeo vectorial proporcionándole diferentes conjuntos de datos de origen (con exactamente la misma dimensionalidad en los vectores de entrada y salida).

En otras palabras, una "red neuronal artificial" no aprende nada y no es una red de neuronas ni nada por el estilo. No entiende nada de los datos y los vectores de salida no tienen ningún significado inherente: su semántica siempre la atribuyen los humanos según sus conocimientos sobre el programa estadístico que han cargado en la máquina de mapeo vectorial.

Como veremos, para programar una máquina de mapeo vectorial (también conocida como RNA), se ejecuta para calcular una salida y luego ajusta automáticamente los vectores paramétricos de sus reductores hacia atrás, desde el último filtro hasta el primero, de modo que, tras cada iteración, su salida se asemeja un poco más a la prevista. Se trata de un proceso iterativo basado en ejemplos cuyas relaciones recurrentes son "absorbidas" por los vectores paramétricos de los reductores vectoriales y luego reproducidas en tiempo de ejecución.

Por eso se trata de programación estadística: se parte de un conjunto de datos de origen (denominado erróneamente "conjunto de entrenamiento" en la jerga de la IA/ML) y, tras un proceso de compilación diseñado específicamente para esa máquina virtual concreta, se obtiene un binario que dicha máquina puede ejecutar.

Aunque dicho binario no está constituido por una secuencia de instrucciones, no deja de ser una transformación algorítmica de la fuente original que, a pesar de expresarse como matrices crípticas, sigue conteniendo todas las características relevantes de las fuentes.

El software ejecutado por una máquina de mapeo vectorial no es un *"modelo"*, ya que no aporta ninguna información sobre las relaciones que absorbió durante su programación estadística. En cambio, es simplemente un software como cualquier otro, que describe un proceso riguroso (aunque desconocido) que una máquina específica debe seguir para calcular de forma automática y determinista (o más bien aproximar) una salida deseada a partir de una entrada dada.

Esto significa, por ejemplo, que el *"modelo"* de Copilot es una obra derivada algorítmica de todos los repositorios públicos subidos a GitHub y, como tal, Microsoft debería cumplir con todo el *network-copyleft* que exige que dicha obra derivada se distribuya a los usuarios.

Una máquina de mapeo vectorial en Python

Para mostrar cómo el *"modelo"* de una "red neuronal artificial" contiene partes sustanciales y valiosas de las fuentes utilizadas para programarla, montamos y programamos un sencillo emulador de una VMM diseñada para realizar una tarea clásica de clasificación: el "reconocimiento" de dígitos escritos a mano de la base de datos MNIST.

El código es extremadamente sencillo, evitando cualquier dependencia externa **excepto** la biblioteca estándar de Python y prescindiendo de optimizaciones matemáticas sofisticadas que harían el proceso más difícil de seguir, pero muestra cómo compilar el conjunto de datos de origen en un binario funcional, cómo comprobar su precisión y cómo descompilarlo.

Aunque se trata de una prueba de concepto de "aprendizaje profundo" bastante sencilla, siempre es posible (aunque costoso) construir un conjunto de compiladores/descompiladores para cualquier sistema de "aprendizaje automático", incluyendo OpenAI y las máquinas de mapeo vectorial específicas que operan el Copilot de GitHub.

Ensamblaje de una máquina de mapeo vectorial

El módulo `vmm.py` describe los componentes básicos de una máquina de mapeo vectorial.

Definimos `VectorMapper` como cualquier dispositivo que convierte un vector de un determinado espacio vectorial (de dimensión `inputSize`) en otro vector de otro espacio (de dimensión `outputSize`).

Un ejemplo es el `VectorReducer`, un dispositivo sencillo que constituye el bloque de construcción básico de las máquinas de mapeo vectorial: es el mítico *"perceptrón"*, también conocido como *"neurona artificial"* en la jerga de la IA/ML.

En aras de la simplicidad, incorpora la clásica función de "transferencia" sigmoide.

El `ParallelVectorMapper` es un mapeador vectorial que compone varios `VectorReducer` y los ejecuta sobre el mismo vector de entrada (potencialmente, en paralelo), devolviendo un vector multidimensional compuesto por sus salidas.

La `VectorMappingMachine` es simplemente una cadena de `ParallelVectorMapper` que puede mapear un vector del espacio de entrada a un vector del espacio de salida *alimentando* a cada filtro (también conocido como "capa" en la jerga de IA/ML) de la cadena el vector de salida devuelto por el anterior.

Una vez que hayamos diseñado nuestra VMM elegida para la tarea, podemos ensamblarla mediante `assemble.py`:

```
$ ./assemble.py classifier-784-32-20-10.vm 784 32 20 10
```

creará una máquina de mapeo vectorial de "aprendizaje profundo" llamada `classifier-784-32-20-10.vm` en tu directorio actual.

Una vez que la máquina virtual esté lista, necesitamos una fuente de muestras para programarla.

Programación de una VMM

Para programar una máquina de mapeo vectorial se necesita un conjunto de datos de origen.

Google, por ejemplo, utiliza los datos personales extraídos de personas que no son conscientes de ello para programar sus VMM de marketing, al igual que Microsoft utilizó el trabajo creativo de montones de desarrolladores que no eran conscientes de ello para crear GitHub Copilot.

En nuestro caso, simplemente utilizaremos el conjunto de datos MNIST.

Una vez que se dispone de un conjunto de datos de origen, a menudo es necesario convertirlo en un conjunto de vectores que se puedan utilizar para programar realmente la máquina: ahí es donde

tienen lugar varios procesos, como la reducción de dimensionalidad, las incrustaciones de palabras (*word embeddings*), etc.

En el ejemplo anterior creamos un "clasificador" capaz de mapear un espacio vectorial de 784 dimensiones a uno de 10 dimensiones: se podría programar con cualquier conjunto de datos de origen que se ajustara a dicha definición (784 características numéricas de entrada y 10 categorías de salida, con codificación *one-hot* para simplificar), pero vamos a utilizar el clásico conjunto de datos MNIST, que consta de 60 000 imágenes en escala de grises de 28x28 píxeles de dígitos escritos a mano.

Dado que la base de datos MNIST se distribuye en un formato binario personalizado y queremos ver cómo recuperar fuentes legibles a partir del *"modelo"* de una "red neuronal artificial", la carpeta `mnist/` contiene un script `mnist2csv.py` que convierte el conjunto de datos incluido en un CSV legible para humanos: los primeros 784 enteros de cada fila corresponden a los píxeles de una imagen rasterizada de 28x28, mientras que el dígito de la última columna corresponde al dígito previsto.

El comando

```
$ ( cd mnist && ./mnist2csv.py )
```

creará `source.csv` y `test.csv` en el directorio `mnist/`.

Se proporciona un script `plot.py` muy sencillo (que depende de matplotlib -*depending on* *`matplotlib`* en el original-) para que puedas inspeccionar el dígito de cada muestra si te apetece.

Compilación de los fuentes

Ahora que tenemos un conjunto de datos de origen que podemos utilizar legalmente para programar nuestro "clasificador de dígitos", solo tenemos que compilarlo en un binario.

El script `compile.py` produce un binario que puede ejecutarse en una Vector Mapping Machine de destino (*target Vector Mapping Machine* en el original, he utilizado la expresión *de destino* por *target* porque no se me ocurre otra mejor) a partir de un conjunto de datos de origen en formato CSV. El formato ejecutable se describe mediante las clases de `ef.py`.

El procedimiento `loadSamples` carga las filas del archivo CSV en una lista de `Sample`, normalizando cada columna y recopilando `SourceStats` útiles, como el valor máximo y mínimo de cada columna de entrada y la lista ordenada de categorías de salida. Dichas estadísticas serán utilizadas en tiempo de ejecución por la VMM para convertir las filas CSV de entrada proporcionadas por los usuarios en vectores de entrada compuestos por `float` en el rango de 0 a 1.

El procedimiento `compile` es donde los vectores paramétricos de cada `VectorReducer` que constituyen la máquina se modifican iterativamente para aproximarse mejor a la salida deseada.

Técnicamente hablando, ejecuta un clásico descenso de gradiente estocástico con retropropagación de errores. En pocas palabras, se itera varias veces (también llamadas *"épocas"*) sobre el conjunto de datos de origen, se intenta ver qué salida produce la VMM para cada muestra, se calcula la diferencia con la salida esperada y luego se retrocede filtro por filtro, modificando ligeramente el

vector paramétrico de cada reductor para que produzca una salida ligeramente mejor dada dicha entrada.

Aquí no hay magia, nadie está "aprendiendo" nada: simplemente estamos cambiando iterativamente los números que cada reductor vectorial multiplica por los elementos del vector de entrada para que produzca una salida más similar a la que queremos obtener.

Como veremos más adelante, vale la pena señalar que el procedimiento `compile` no descarta **toda**s las informaciónes *(does **not** throw away all informations*) sobre cómo cada muestra influye en el ejecutable final, sino que **registra algunos datos** (logs) durante la última *"época"*.

Así que para compilar el conjunto de datos de origen solo tienes que ejecutar

```
$ ./compile.py classifier-784-32-20-10.vm mnist/source.csv mnist/digit-reader.bin
```

Al cabo de un rato (puede tardar unas horas en máquinas lentas), obtendrás tu programa en `mnist/digit-reader.bin`.

A continuación, puedes comprobar la precisión del programa ejecutando

```
$ ./test.py classifier-784-32-20-10.vm mnist/digit-reader.bin mnist/test.csv
```

Predicciones correctas: 9530/10000 (95,3 %)

¿Ha "aprendido" nuestra "red neuronal" de "aprendizaje profundo" a "leer" dígitos manuscritos?

No

Simplemente hemos programado estadísticamente una máquina de mapeo vectorial para plegar un vector de 784 dimensiones en un vector de 10 dimensiones, y resulta que, de entre todas las infinitas reducciones posibles que podría realizar, una de ellas coincide con algunas regularidades observadas en el conjunto de datos MNIST.

De hecho, esas regularidades características son estadísticamente relevantes en el conjunto de datos de origen, por lo que se "absorben" en los *pesos* de los reductores que componen la máquina durante la compilación.

Estos son aspectos fundamentales que hay que comprender y recordar al pensar en este tipo de tecnologías.

En primer lugar, la VMM no aprende nada: siempre reproduce una reducción vectorial que es estadísticamente relevante en el conjunto de datos de origen, simplemente porque la hemos programado estadísticamente para que lo haga. En este caso, dicha reducción es posible porque los seres humanos diseñaron un conjunto regular de 10 signos para comunicar dígitos y el conjunto de datos de origen MNIST contiene 60 000 de dichos signos dibujados por seres humanos para seres humanos. Para comunicarse con otros seres humanos, estos tuvieron que reproducir signos similares a los que ellos entenderían, y esto generó las regularidades que pueden ser "absorbidas" y reproducidas por una máquina de mapeo vectorial.

En otras palabras, aquí no hay inteligencia en juego, salvo la humana.

En segundo lugar, las regularidades "absorbidas" por los *pesos* de los reductores vectoriales son aquellas que caracterizan el trabajo creativo realizado por los seres humanos: el trabajo creativo que produjo los símbolos que utilizamos como dígitos numéricos y el trabajo creativo de las personas que realmente dibujaron esos dígitos y el trabajo de las personas que los recopilaron en el conjunto de datos de origen. Podemos utilizar legalmente estas obras por diferentes motivos, pero, al fin y al cabo, todos esos motivos se basan en el consentimiento de las personas que contribuyeron a ellas.

Esto es relevante porque, siempre que *el conjunto de datos de origen* (*the source dataset*) utilizado para programar una VMM no esté compuesto por datos emitidos por objetos u objetos observados (como datos personales o datos sobre un fenómeno físico), el ejecutable producido durante la compilación es siempre una obra derivada del conjunto de datos de origen, al igual que un binario es una obra derivada del código fuente utilizado para compilarlo.

Además, en este segundo caso, el programa ejecutado por una VMM/RNA incorpora exactamente aquellas características del conjunto de datos de origen que constituyen el valor creativo de la obra.

Descompilación de (el software ejecutado por) una "red neuronal artificial"

ERRATA CORRIGE:

> El proceso que se describe a continuación no prueba ni refuta nada.

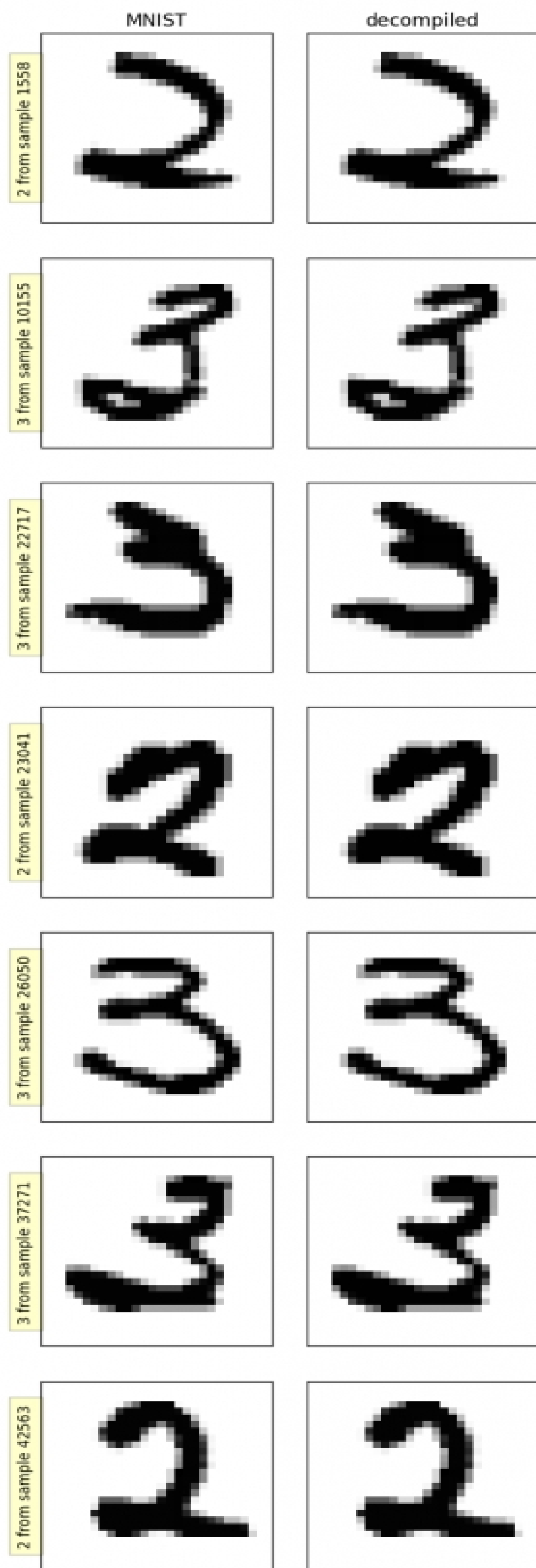
Para demostrar que una "red neuronal artificial" contiene el conjunto de datos de origen utilizado para "entrenarla" de forma críptica (pero no cifrada), descompilaremos el ejecutable generado y recuperaremos el conjunto de datos de origen con un simple comando:

```
$ ./decompile.py mnist/digit-reader.bin mnist/digit-reader.sources.csv
```

Esto creará un nuevo archivo CSV llamado `digit-reader.sources.csv` que se puede comparar con el conjunto de datos MNIST en `mnist/sources.csv`.

En nuestro ejemplo, 59 993 muestras descompiladas coinciden perfectamente con las del conjunto de datos MNIST original (~99,99 %), y solo siete imágenes no coinciden perfectamente con las originales, pero las diferencias son imperceptibles a simple vista.

De hecho, en esas imágenes, las ligeras variaciones en la escala de grises de algunos píxeles se deben a errores de redondeo durante las operaciones de coma flotante (*floating point operations*). Dichos errores aumentan con el incremento de la "precisión de predicción" del ejecutable, pero son solo problemas de implementación que podrían solucionarse utilizando bibliotecas de aritmética de coma flotante de precisión arbitraria (*arbitrary precision floating-point arithmetic libraries*).



Así pues, podemos ver que el programa ejecutado por la "red neuronal artificial" contiene en realidad todo el conjunto de datos de "entrenamiento", a pesar de estar organizado en un conjunto bastante críptico de matrices numéricas.

La mayor parte del proceso de descompilación tiene lugar en el procedimiento `decompile`, que recorre los registros grabados durante la última iteración sobre el conjunto de datos de origen (la última "época" en la jerga de IA/ML) y revierte el proceso de compilación.

Para cada línea de registro, se recorre la VMM hacia atrás, desde el último filtro hasta el primero.

Para cada filtro utilizamos la salida registrada, el error y las variaciones de los *pesos* de un nodo centinela para calcular el vector de entrada recibido del filtro anterior. A continuación, calculamos las variaciones de peso para todos los demás reductores del filtro y las aplicamos.

Ahora bien, si te gusta demasiado la narrativa de la *IA*, si adoras la futurología o si eres un transhumanista asustado por imaginarios "riesgos existenciales", deberías estar seriamente indignado: ¡la "red neuronal" está "desaprendiendo"!

¡Estamos literalmente "lavando el cerebro" a la máquina!

¡También podría pasarles a tus imaginarios herederos simulados!

¡Y no hay nada que puedas hacer al respecto! :-PP

Hablando en serio, cuando todos los reductores de los filtros se han restablecido a los parámetros anteriores, calculamos los errores del filtro anterior y luego repetimos el proceso con él.

Finalmente, cuando se han procesado todos los filtros, la entrada del primer filtro se reescala y se guarda en el archivo CSV de salida, y pasamos al siguiente registro.

¿Cuál es el truco?

Normalmente, durante el proceso de compilación (también conocido como "entrenamiento"), cada muestra del conjunto de datos de origen se convierte en una miríada de vectores numéricos que son "absorbidos" por los vectores paramétricos de los reductores de la máquina... y luego son descartados.

Al registrar muy pocos de esos vectores (uno por cada filtro, para ser precisos) y revertir el proceso de compilación, podemos calcular todos los demás vectores y la entrada original.

En concreto, para cada muestra compilada durante la última iteración sobre el conjunto de datos, el ejecutable conservará:

el vector de salida producido por la VMM
el vector de error (la diferencia entre la salida calculada y la proporcionada por las fuentes de la muestra)
la variación de los *pesos* aplicada a un único `VectorReducer` para cada `ParallelVectorMapper` del programa.

No importa cuán compleja o grande sea una "red neuronal", basta con los cambios en el vector paramétrico de una sola "neurona" de cada "capa".

Tenga en cuenta que dichos vectores no son suficientes, por sí solos, para recuperar el conjunto de datos original: los *pesos* finales utilizados por los reductores que componen la máquina también son

necesarios durante el proceso de descompilación, ya que contienen proyecciones fundamentales del conjunto de datos (*dataset*).

En otras palabras, cualquier "modelo" de una "IA" es una parte sustancial de una obra derivada más amplia basada en el conjunto de datos utilizado durante el "entrenamiento".

Descartar el resto de la obra (los registros que `compile.py` conservó) es solo una forma de ocultar las pruebas de dicha derivación.

Por eso a nadie debería sorprenderle que GitHub Copilot sugiera incluir código con *copyleft* en software propietario: ¡literalmente ha sido "entrenado" para violar los derechos de autor de los hackers desde el principio!

De hecho, sigue conteniendo partes sustanciales del código con *copyleft* subido a GitHub, pero no cumple con las licencias de origen.

Conclusiones

> **NOTA:** el proceso de descompilación descrito anteriormente presentaba graves deficiencias, por lo que no prueba ni refuta nada sobre el "modelo".
>
> Sin embargo, aún se pueden derivar las siguientes conclusiones de la separación adecuada entre la ingeniería de las máquinas de mapeo vectorial y el proceso necesario para programar estadísticamente sus reductores vectoriales, tal y como se ha descrito anteriormente.

Probablemente, "GitHub Copilot" debería renombrarse como ***"Microsoft Copy(a)lot"***. ;-)

Su software, al igual que cualquier otro "modelo" producido mediante esas técnicas que actualmente se engloban bajo el término "inteligencia artificial", es una obra derivada del conjunto de datos de origen (*source dataset*), al igual que cualquier binario producido por un compilador es una obra derivada de las fuentes (*sources*).

Por lo tanto, Microsoft debería obtener una licencia de los autores de las fuentes (sources) que compiló en dicho software, o cumplir con todas sus licencias.

Por otro lado, si Microsoft tiene razón y programar una máquina de mapeo vectorial es suficiente para eliminar los derechos de autor de una obra, podríamos utilizar la técnica mostrada aquí para compilar textos, canciones, películas e incluso software propietario en un programa para una VMM estándar y luego descompilarlo cuando sea necesario, para reproducirlo o ejecutarlo: la "inteligencia artificial" eliminaría cualquier riesgo legal para las personas que distribuyen el binario compilado.

De hecho, si lo que hizo Microsoft fuera realmente "uso legítimo", cualquier obra protegida por derechos de autor podría utilizarse de la misma manera. En otras palabras, si Microsoft tiene razón, habría dejado obsoletas la mayoría de las leyes oscurantistas de "propiedad intelectual" que plagan nuestra sociedad cibernética: un enorme paso adelante que todos deberíamos acoger con gratitud.

Investigación adicional

ERRATA CORRIGE:

> El proceso de descompilación descrito anteriormente tenía graves deficiencias.
>
> Como consecuencia, la única forma de garantizar un mínimo de transparencia sobre el proceso de compilación de una "red neuronal artificial" es conservar todos los datos necesarios para replicar exactamente el proceso en sí, lo que incluye no solo el conjunto de datos de origen, sino también los *pesos* iniciales asignados a cada reductor de vectores, el orden de los ejemplos (*samples*, -también se puede traducir por muestras-) durante cada "época", la entrada obtenida por cualquier fuente de entropía, el peso de los ejemplos (*samples*) (también conocido como "tasa de aprendizaje") y cualquier otra decisión de ingeniería que pudiera afectar al estado final de la máquina... para cada máquina y cada programa probado durante la validación cruzada.
>
> Solo con todos esos datos podrás, obviamente, verificar si una determinada muestra (*o ejemplo, como he traducido antes*)se utilizó correctamente durante la compilación o eliminar muestras problemáticas del software ejecutado por la máquina final simplemente recompilando el conjunto de datos de origen sin ellas, mientras mantienes todo lo demás sin cambios.

La técnica que aquí se muestra es bastante sencilla, pero puede adaptarse a cualquier otra técnica de "aprendizaje automático". Solo hay que no descartar (*to **not throw away***) datos relevantes durante la compilación del conjunto de datos de origen.

Además, dicha técnica podría ampliarse para verificar, en condiciones adecuadas, si se ha utilizado un determinado conjunto de muestras durante la compilación de una "inteligencia artificial" concreta durante el análisis forense, y de qué manera.

Por último, si los datos requeridos se han conservado debidamente durante la realización de un sistema de "aprendizaje automático", dicha técnica podría adaptarse para eliminar muestras problemáticas del programa ejecutado desde el sistema.

Licencia

Este ensayo y todo el código Python distribuido con él se te conceden bajo los términos de la Licencia Hacking (véase `HACK.txt`).

El conjunto de datos MNIST se ha copiado en el repositorio tal y como lo solicitaron sus mantenedores.

Hasta aquí, el artículo de Tesio. Mi traducción tendrá errores, seguro, así que recomiendo que cada cual se enfrente a las fuentes y cometa su propios errores. Y si alguien me corrige los míos, pues sería todo un lujo.