Trabajo Final Integrador
Autoencoder Convolucional

José Ignacio Robledo

22 de julio de 2021

Introducción

El objetivo de este trabajo integrador es realizar un autoencoder convolucional para aprender la función identidad utilizando la base de datos provista por MNIST de dígitos escritos a mano y digitalizados. Un autoencoder es un modelo generativo. Este tipo de red aprende una nueva representación de los datos de entrenamiento y puede ser utilizado para la generación de nuevas instancias de los datos (idea sobre la cual volveré en los resultados). El requisito para esta red neuronal será que consista en 784 unidades de entrada y 784 neuronas de salida. Esto se debe a que las imágenes de MNIST consisten en $28 \times 28 = 784$ pixeles. Como el objetivo es obtener la función identidad, entonces la capa de salida debe tener la misma dimensión que la capa de entrada.

En el práctico 3, entrené una red feed-forward auto-encoder, en donde utilicé la función de pérdida del error cuadrático medio ya que resulta una buena medida para definir una distancia entre dos imágenes (la de entrada y la de salida). El objetivo de entrenar la red es que esta medida de distancia sea lo más cercana a cero posible. En ese práctico además utilicé el método de optimización Adam para buscar el mínimo global de la función de pérdida, implementando un dropout del 10 % ($p = 0{,}1$) y utilizando minibatch. Como el segundo objetivo de este trabajo integrador es comparar los resultados obtenidos con el auto-encoder del práctico 3 y el convolucional, utilizaré la misma función de pérdida y el mismo optimizador. Además, evaluaré distintas arquitecturas para la red neuronal convolucional.

En cuanto al método de optimización, Adam es un acrónimo para *Adaptative Moment Estimation* y viene a ser una variación del método RMSProp (*Root Mean Square Propagation*). En Adam, no sólo se realiza un promedio móvil de las magnitudes de los últimos gradientes calculados como en RMSProp, sino que también se realiza un promedio móvil del segundo momento de los gradientes y ambas cantidades son tenidas en cuenta para elegir hacia donde moverse en el espacio de los parámetros. Además, en el método Adam se utiliza una tasa de aprendizaje distinta para cada parámetro la cual es adaptativa. Es decir, los parámetros que recibirían normalmente actualizaciones pequeñas o menos frecuentes van a recibir actualizaciones mayores con Adam y viceversa.

Componentes de la red

Un autoencoder contiene dos componentes:

- Un encoder o codificador que toma una imagen de entrada y devuelve una representación de la imagen original de menor dimensionalidad;
- Un decoder o decodificador que toma la representación de menor dimensión y reconstruye la imagen original.

A continuación haré una breve descripción de las componentes que utilizaré para armar el encoder y el decoder y que son comúnmente utilizadas en las redenes neuronales convolucionales.

Capa convolucional: Las capas convolucionales aplican varios filtros o kernels a la imagen de entrada, con el objetivo de encontrar y resaltar las características principales de la misma. La aplicación de estos filtros se realiza mediante la operación de correlación cruzada, es decir, barriendo el kernel a lo largo de la imagen. Cada capa convolucional tiene dos tipos de parámetros: los pesos y los sesgos. El número total de parámetros es la suma de los pesos y los sesgos. El número de pesos de la capa convolucional W_c se calcula a partir del tamaño del kernel K, del número de canales de la imagen de entrada (C) y del número de kernels elegido (N) como

$$W_c = K^2 \times C \times N. \tag{1}$$

Hay un sesgo B_c introducido por cada kernel, con lo cual $B_c = N$. Así el número de parámetros de la capa convolucional es

$$P_c = W_c + B_c. \tag{2}$$

Para mejorar la intuición respecto a cómo funciona este tipo de capa, mostraré el efecto de una única capa convolucional que aplica ocho kernels aleatorios de tamaño 3×3 a una imagen de entrada. Fijé un canal de entrada ya que las imágenes MNIST son en escala de grises. Las capas convolucionales pueden reducir la dimensión de la imagen de entrada (down sampling) mediante el parámetro "*stride*". Este parámetro asigna la cantidad de pixeles que el algoritmo debe mover al kernel entre dos aplicaciones sucesivas del mismo. Si el stride es 1, entonces se va moviendo de a un pixel a la vez, aplicando en cada caso los 8 kernels aleatoriamente elegidos, resultando así en 8 imágenes de la misma dimensión que la original. Si el stride=2, el kernel se aplica cada dos pixeles. En este caso, las sucesivas aplicaciones del kernel terminarán reduciendo la dimensión de la imagen a la mitad. Puede suceder que exista un problema al llegar al final de una fila o columna de pixeles debido a que el kernel utilizado es de dimensión 3×3, con lo cual podrían faltar filas o columnas para poder aplicar correctamente el kernel y reducir la imagen al tamaño deseado (14×14 si la imagen original es de 28×28). Esto se soluciona agregando una determinada cantidad de filas o columnas de ceros, comúnmente denominado "*padding*" (padding=1). El resultado de una imagen al pasar por una capa convolucional con 1 canal de entrada, 8 de salida, stride=2 y padding=1 puede verse en la fig. 1. La imagen origianl se muestra a la izquierda, de dimensión 28×28. En este caso, las 8 imágenes de salida son de dimensión 14×14. Puede observarse como la capa convolucional, además de reducir la dimensión, resalta distintas características de la imagen original mediante los distintos kernels. Este conjunto de 8 imágenes de 14×14 se almacenan en un tensor de dimensión $8 \times 14 \times 14$.

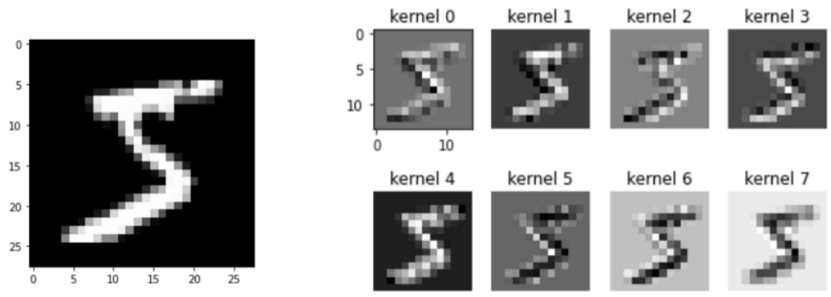

Figura 1: La imagen grande a la izquierda es una imagen aleatoria del conjunto de imágenes de entrenamiento de la base MNIST. Las ocho imágenes a la dereche son el resultado de los 8 kernels arbitrarios elegidos por la capa convolucional con stride propuesta. Cada dimensión de la imagen original se ve reducida a la mitad.

Max pooling: Otra manera en la que se podría haber reducido dimensionalidad del problema y por ende la cantidad de parámetros a ajustar, sería mediante el uso del concepto de "Max pooling". Consiste en seleccionar el máximo de cada subconjunto de pixeles $p \times p$ de la imagen, reduciendo así una imagen $n \times n$ a una imagen $(n/p) \times (n/p)$. A modo ilustrativo, generé otra red neuronal que aplica una convolución con stride=1, manteniendo la dimensión original 28×28, y luego aplica una capa max pool bi-dimensional con $p = 2$. El resultado puede verse en la fig. 2. Las imágenes resultantes tienen la dimensión 14×14 esperada.

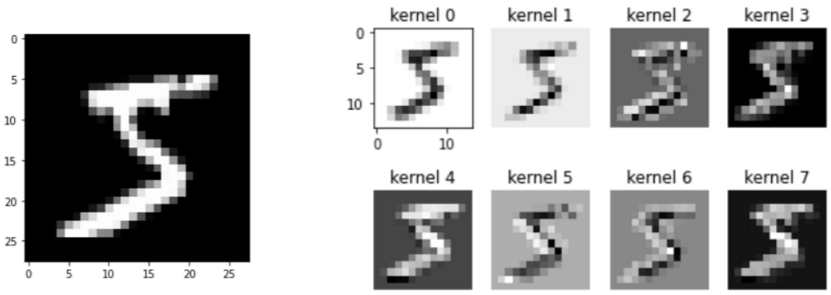

Figura 2: Resultados de aplicar pooling 2x2 a las 16 imágenes resultantes de la primera capa convolucional sin stride. El tamaño de las imágenes resultantes es de 14x14 pixeles.

Como primera observación, vemos que incluir una capa convolucional sin stride en la arquitectura seguida de una capa max pooling tiene el mismo efecto de reducción de dimensión de la imagen original que aplicar una capa convolucional con el parámetro de stride seleccionado adecuadamente. La ventaja de incluir una capa de Max Pooling es que viene "*sin costo*" ya que no agrega parámetros nuevos a ajustar en la red. Sin embargo, de este modo estamos eligiendo de antemano cómo seleccionar el pixel conveniente para la representación de menor dimensión de la imagen (le estamos exigiendo que elija el máximo). Del otro modo, al poner una capa convolucional con parámetro de stride, en cierto sentido estamos permitiendo que la red neuronal *aprenda* el pooling conveniente que lleva a la imagen final de menor dimensión. El efecto del max pooling termina siendo reemplazado por una convolución de los pixeles con el kernel elegido. Dado que puedo elegir la arquitectura de la red, optaré por utilizar sólo la capa convolucional con stride. Sin embargo haré una pequeña comparación entre estas dos posibilidades en los resultados.

Capa ReLU: El acrónimo viene del inglés "*Rectified Linear Unit*". Esta capa se encarga de devolver el valor de entrada si éste es mayor que cero y devolver cero en caso contrario. Facilita en gran medida el entrenamiento de la red neuronal y ayuda normalmente a obtener mejor performance. Como la función es lineal para valores mayores que cero, tiene las propiedades deseables de una función lineal de activación al entrenar la red utilizando backpropagation. Su derivada es fácil de calcular y permite una representación esparsa de los parámetros, lo cual permite la aceleración del aprendizaje y la simplificación del modelo.

Arquitectura de la red

En esta sección explicaré la arquitectura de la red más sencilla que se me ocurrió, tratando de disminuir el número de parámetros a entrenar. Luego en la instancia oral podemos discutir las otras opciones que estuve pensando también, con arquitecturas más complejas (que pienso innecesarias para este problema).

El codificador o encoder consistirá en la siguiente secuencia de capas:

1. Una capa convolucional bi-dimensional que tome de entrada la imagen de $28 \times 28 = 768$ pixeles y devuelva un tensor de dimensión $8 \times 9 \times 9$ (8 canales, 9 pixeles en x y 9 pixeles

en y). Para esto, utilizará kernels de dimensión 3×3, con $stride = 3$ y $padding = 0$. Utilizando las ecuaciones 1 y 2, vemos que esta capa incluirá 80 parámetros.

2. Una capa ReLU.

3. una segunda capa convolucional bi-dimensional, que tome de entrada el tensor $8 \times 9 \times 9$ proveniente de la capa ReLU y devuelva un tensor de $16 \times 3 \times 3$. Esta capa agregará 1168 parámetros.

El decodificador o decoder consistirá en la secuencia:

1. Una capa convolucional transpuesta 2D que tome la imagen del espacio embedido, codificada por el encoder, y devuelva un tensor de $8 \times 9 \times 9$, utilizando kernels de 3×3, con $stride = 3$. Esta capa agrega 1160 parámetros.

2. Una capa ReLU.

3. Una seguna capa convolucional transpuesta 2D que toma de entrada la imagen proveniente de la capa ReLU y devuelve un tensor de dimensión $1 \times 28 \times 28$ (equivalente a una imagen de 28×28), utilizando kernels de 3×3 y $stride = 3$ (más un out padding de 1 para obtener la dimensión correcta). Agrega 73 parámetros.

4. una capa sigmoidea para dejar los valores entre 0 y 1 y así utilizar la escala de grises al igual que MNIST.

El número total de parámetros a aprender es 2481.

Siguiendo el mismo razonamiento, se planteó una segunda arquitectura con tres capas convolucionales en vez de dos, que se describirá de forma más sintética, la cual presenta gran cantidad de parámetros (11748). Esta red claramente aprende mejor la función identidad ya que tiene disponible muchos más parámetros. Pero desde mi punto de vista estadístico, he optado por un compromiso entre bondad de ajuste y cantidad de parámetros, decidiendo que la red con menor cantidad de parámetros es mejor modelo por ser más parsimonioso (ya que ambas llegan a resultados razonables).

La descripción sintética de la segunda red es:

Encoder:

1. capa Conv2D(canalEntrada=1,canalSalida=8,kernel=3×3,stride=2,padding=1) (salida de dimensión 14×14)

2. capa ReLU

3. capa Conv2D(canalEntrada=8,canalSalida=16,kernel,3×3,stride=2,padding=1) (salida de dimensión 7×7)

4. capa ReLU

5. capa Conv2D(canalEntrada=16,canalSalida=32,kernel,3×3,stride=2,padding=1) (salida de dimensión 3×3)

Decoder:

1. capa ConvTranspuesta2D(canalEntrada=32,canalSalida=16,kernel=3×3,stride=2) (salida de dimensión 7×7)

2. capa ReLU
3. capa ConvTranspuesta2D(canalEntrada=16, canalSalida=8,kernel=3×3, stride=2, padding=1, outPadding=1) (salida de dimensión 14×14)
4. capa ReLU
5. capa ConvTranspuesta2D(canalEntrada=8, canalSalida=1,kernel=3×3, stride=2, padding=1, outPadding=1) (salida de dimensión 28×28)
6. capa sigmoidea

Entrenamiento y prueba

Para el entrenamiento se utilizaron las 60000 imágenes del conjunto de entrenamiento de la base MNIST. Se entrenó durante 20 épocas, utilizando un minibatch de 64. El método Adam de optimización se utilizó con una tasa de aprendizaje de 0,001 y un peso de decaimiento de 0,00001.

Luego de cada batch se evaluó el resultado de la función de pérdida MSE en las imágenes del conjunto de entrenamiento y en el de prueba (10000 imágenes) para corroborar que no haya sobreajuste a los datos.

Resultados

Sólo a modo ilustrativo y tratando de entender cómo evoluciona el aprendizaje de la red, almacené los resultados intermedios del entrenamiento dentro de la primera época, luego de haber corrido 20 minibatches y 40 minibatches. Tomé en estos instantes ya que esta red aprende muy rápido y en instantes posteriores las imágenes son muy similares a simple vista. Los resultados se pueden ver en la fig. 3. En esta imagen se observan 9 imágenes al azar del conjunto de entrenamiento en la fila denominada *original* y luego inmediátamente debajo se observa el resultado de pasar por el autoencoder luego del entrenamiento durante distinta cantidad de minibatches dentro de la misma época (sin haber pasado por todo el conjunto de datos todavía!). Vemos que inicialmente los pesos están asignados arbitrariamente, por eso las imágenes del guess inicial luego de pasar por el autoencoder convolucional sin entrenar son ruidosas. A medida que avanza el entrenamiento (medido en cantidad de minibatches) se observa una mejora sustancial en la imagen de salida de la red.

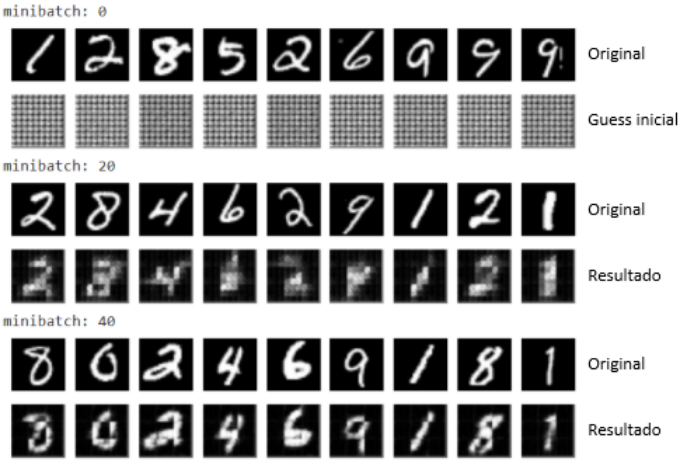

Figura 3: Evaluación del modelo en distintas instancias del entrenamiento dentro de la primera época.

Comparación entre las tres arquitecturas propuestas

En la fig. 4 comparo la red cuyo encoder consiste en 2 capas convolucionales con stride=3 (primera arquitectura propuesta, color verde) con la red cuyo encoder presenta 3 capas convolucionales con stride=3 (color azul). Además comparo con una red donde se modifica el encoder, cambiando las capas convolucionales con stride=3 por capas convolucionales sin stride (stride=1) y un posterior max pooling de 3×3 (color rojo) luego de cada capa convolucional.

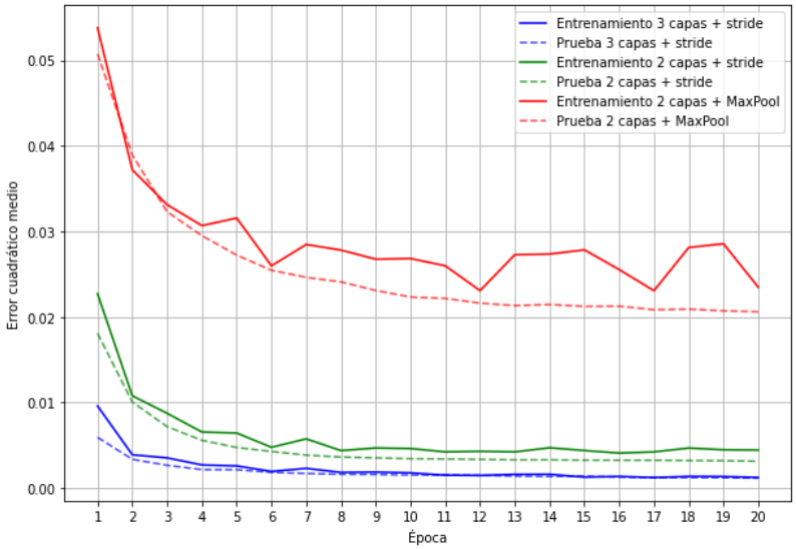

Figura 4: Comparación del error cuadrático medio para las dos arquitecturas propuestas (verde y azul) y la variante de la red más sencilla (verde) con max pooling en vez de stride (rojo).

Volviendo al comentario que había hecho anteriormente, en esta base de datos resulta ser más eficiente desde el punto de vista de la minimización del error cuadrático medio el hecho de permitir que la red aprenda la información que nosotros estamos poniendo a priori en la capa de max pooling (elegir el máximo pixel en vez de alguna combinación lineal, o convolución por ej.). Además observamos que la red que minimiza el ECM es la red con mayor cantidad de parámetros, que tiene 2 capas ocultas. Esto tiene sentido también ya que la reducción de dimensión que se da es menos drástica que en el caso de tener sólo una capa oculta. Sin embargo, si volvemos a la cantidad de parámetro, estamos intercambiando disminuir el error cuadrático medio a la mitad aproximadamente por agregar unos 9000 parámetros (más del 350 % de parámetros de la red mas parsimoniosa!). Minimizar el error cuadrático medio no lo es todo. Si observamos los resultados de salida de cada una de las redes, podemos ver que "a simple vista" no hay gran diferencia entre los resultados obtenidos y ambas funciones identidad ajustadas se asemejan muy bien a la función identidad. Esto refuerza la elección que hago por la arquitectura más simple.

Comparación con el práctico 3

En el caso del práctico 3 habíamos utilizado la misma función de pérdida, el error cuadrático medio (ECM). La red neuronal que habíamos entrenado era bastante más sencilla (menor cantidad de parámetros a ajustar), con lo cual era lógico de esperar que las arquitecturas propuestas para el autoencoder convolucional aprendan más rápido la función identidad. Efectivamente, si observamos la figura 5, que muestra la figura 6 del práctico 3, donde se ve la variación del error cuadrático medio en función de las épocas para el auto-encoder feed-forward, y la comparamos con la figura 4 de este práctico, vemos que en los casos planteados (salvo en el de max pooling) el autoencoder convolucional aprende mejor y más rápido.

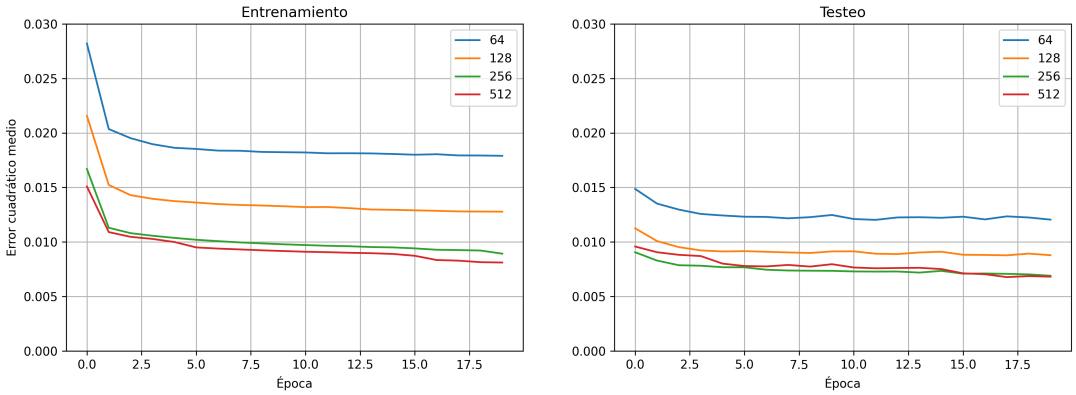

Figura 5: resultados obtenidos en el práctico 3, para una red neuronal feed-forward autoencoder con una capa oculta de distinta dimensión (64, 128, 256 y 512).

A usar el autoencoder convolucional!

Lo que más interesante me resulta del autoencoder convolucional es el espacio latente, el espacio embebido, donde queda codificada la información de la imagen original. A continuación exploro el espacio latente del autoencoder convolucional entrenado con dos capas convolucionales y stride en vez de max pooling. Una vez entrenada la red, podemos pasar una imagen solo por el encoder o codificador y observar el resultado. La figura 6 muestra el resultado del espacio latente para la imagen original de un número nueve (imagen de la izquierda). Debido a la arquitectura de esta red, el codificador devuelve un tensor de dimensión $16 \times 3 \times 3$. Es decir, 16 canales (uno por cada kernel) y cada resultado de aplicar los kernels 3×3, justo de de dimensión resultante 3×3 también (ojo, no estoy mostrando los kernels, sino los resultados de aplicar los kernels a las imágenes 9×9 de la capa oculta). Este resultado se muestra en la grilla 4×4 ubicada al centro de la figura 6. Podemos ver que de aquí no resulta sencillo entender que se trata del número nueve. Sin embargo, allí se encuentra toda la información necesaria para realizar una buena reconstrucción de la imagen original. Para esto es necesario el decodidicador que entrenamos que recibe de entrada este tensor $9 \times 3 \times 3$ y que nos devuelve una reconstrucción fidedigna de la imagen original (imagen a la derecha titulada *Recon*).

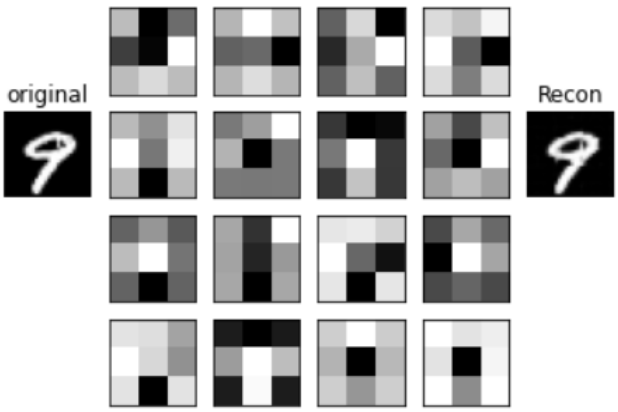

Figura 6: La imagen de la izquierda titulada original muestra la imagen seleccionada del conjunto de datos MNIST de prueba. La grilla de 16 imágenes, cada una de dimensión 3×3, en el centro muestra la representación de la imagen original en el espacio latente obtenida al pasar por el encoder. La imagen de la derecha muestra la reconstrucción obtenida al introducir la representación del espacio latente en el decodificador del autoencoder convolucional.

Una aplicación interesante del autoencoder es la de generar nuevas imágenes a partir de la exploración de este espacio embebido (es decir, su uso como modelo generativo). Ya que tenemos acceso a este espacio latente, podemos ir al mismo e identificar en él la representación de dos imágenes de la base de datos de prueba, sean x_1 e x_2. Luego podemos realizar distintas combinaciones lineales, por ejemplo de la forma

$$x_i' = \frac{i}{10}x_1 + \frac{(10-i)}{10}x_2, \tag{3}$$

con $i = 1, ..., 10$. Una vez obtenidas estos nuevos puntos del espacio latente, podemos pasarlos por el decodificador, obteniendo nuevas imágenes que no pertenecen ni al conjunto de datos de entrenamiento, ni al de prueba! Muestro un ejemplo de esto en la figura 7, en donde se han tomado dos imágenes del conjunto de prueba, un siete y un cuatro (que se muestran en la primera fila) y se han interpolado (segunda fila). Resulta inclusive más interesante cuando se toman dos imágenes distintas correspondientes al mismo número, como por ejemplo dos imágnes de nueve distintas (fila 3), ya que todos las interpolaciones intermedias parecen números nueve nuevos (fila 4). Estos podrían ser utilizados para aumentar el tamaño de la base de datos o bien remplazar imágenes o corroborar la robustez de la red.

Figura 7: Interpolación obtenida usando la ecuación 3. En las filas con sólo dos imágenes se muestran las originales, obtenidas del conjunto de datos de prueba de la base MNIST. En las con 10 imágenes se muestras las interpolaciones de las imágenes superiores.