Redes Neuronales - Práctico 3: Autoencoder sobre Fashion-MNIST.

Reinaldo Magallanes Saunders*
Facultad de Matemática, Astronomía, Física y Computación, Universidad Nacional de Córdoba
M.Allende y H. de la Torre - Ciudad Universitaria, X5016LAE - Córdoba, Argentina
(Dated: 5 de Agosto de 2022)

I. INTRODUCCIÓN

Uno de los objetivos en el desarrollo de la inteligencia artificial es la creación de sistemas con la habilidad de adquirir su propio conocimiento, mediante la extracción de patrones a partir de datos sin procesar, con el objetivo de realizar predicciones o tomar decisiones sin haber sido explícitamente programados para hacerlo. El estudio de sistemas con esta capacidad, y los algoritmos que la hacen posible, se conoce como *machine learning*. Debido a la diversidad de aplicaciones, existe un gran interés en el desarrollo de sus muchas implementaciones. En particular, la disponibilidad de datos en masa y el incremento del poder computacional producto de avances tecnológicos hicieron posible la proliferación de las redes neuronales.

El desempeño de las redes neuronales, y de los algoritmos de machine learning en general, depende fuertemente de la representación de los datos suministrados. Cada elemento de información incluido en la representación se conoce como característica o *feature*. Muchas tareas de inteligencia artificial pueden ser resueltas mediante la extracción de los features apropiados para esa tarea, y luego suministrarle estos features al algoritmo de machine learning. Sin embargo, para muchas tareas, es dificultoso saber qué features son los que deben ser extraídos. Una posible solución a este problema es utilizar machine learning, no solo para encontrar la función que mapea la representación a la salida deseada, sino también para encontrar la representación apropiada, mediante extracción de características o *feature extraction*. Este enfoque se conoce como representation learning o *feature learning*. El ejemplo típico de un algoritmo de representation learning es un tipo particular de red neuronal conocido como *autoencoder*.

II. REDES NEURONALES

Las redes neuronales son un modelo computacional inspirado por la estructura neuronal del cerebro y consisten en un conjunto de unidades de cómputo simples llamadas *neuronas*, que se encuentran conectadas entre ellas mediante conexiones de sensibilidad variable, denominadas *pesos*. Las *neuronas de entrada* son aquellas que reciben los datos desde el exterior, mientras que las que envían información fuera de la red se denominan *neuronas de salida*.

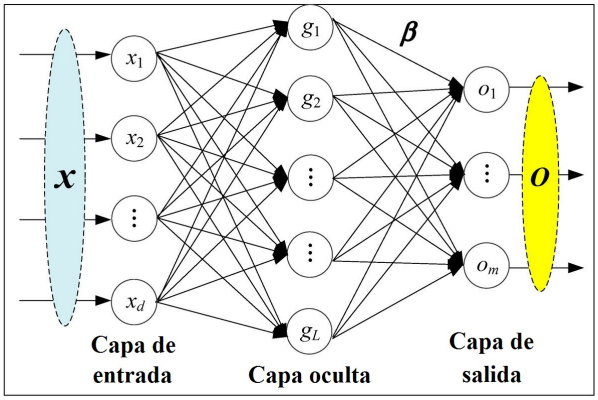

Figura 1: Representación de una red neuronal feedforward de tres capas con una entrada d-dimensional $\boldsymbol{x}$, una capa oculta con L neuronas, una salida m-dimensional $\boldsymbol{O}$, y un tensor de pesos $\boldsymbol{\beta}$. Adaptación[1].

En función del recorrido que sigue la información dentro de la red, las redes neuronales se clasifican en dos grandes tipos: *recurrentes* y alimentadas hacia adelante o *feedforward*. En las redes recurrentes, las neuronas pueden estar conectadas con cualquier neurona de la red, incluso consigo mismas.

Las redes feedforward, en cambio, se organizan en capas de forma que las neuronas de una capa solo pueden recibir información de capas anteriores y enviar información a capas posteriores, y no existen conexiones entre neuronas de una misma capa. La capa formada por las neuronas de entrada se denomina *capa de entrada* y la *capa de salida* es la formada por las neurona de salida, mientras que cualquier capa entre estas dos recibe el nombre de *capa oculta*. Si toda neurona esta conectada a todas las neuronas de las capas anterior y posterior, se dice que la red feedforward es totalmente conectada.

La estructura de una red neuronal se denomina *arquitectura* de la red, y suele representarse en forma de grafo ponderado dirigido donde las neuronas se representan como nodos y las conexiones como aristas dirigidas. En la Figura 1[1] se muestra una representación de una red feedforward de tres capas.

La *regla de propagación* es la forma en que cada neurona, salvo por las de la capa de entrada, interactúa con la información recibida y produce un resultado, combinando en un único valor los resultados de las neuronas de la capa anterior junto con los pesos de las conexiones corres-

*Correo electrónico: rei.magallanes@mi.unc.edu.ar

pondientes. Normalmente, consiste en una combinación lineal de la forma

$$h_j^{(i)} = \sum_k \beta_{j,k}^{(i-1)} s_k^{(i-1)}$$

donde $h_j^{(i)}$ es el resultado de la regla de propagación para la j-ésima neurona de la i-ésima capa, $\beta_{j,k}^{(i-1)}$ es el peso de la conexión que va de la k-ésima neurona de la capa anterior hacia la j-ésima neurona de la i-ésima capa y $s_k^{(i-1)}$ es la salida de la k-ésima neurona de la capa anterior. La salida de la neurona, $s_j^{(i)}$, se calcula aplicando una *función de activación* g a $h_j^{(i)}$. En el caso particular de la capa de entrada, $h_j^{(1)}$ esta dada por la entrada suministrada y la función de activación es la identidad.

A. Aprendizaje

Existen dos grandes categorías de aprendizaje. El *aprendizaje supervisado* usa un conjunto de entrenamiento $\{\boldsymbol{\xi}^{(\alpha)}, \boldsymbol{\zeta}^{(\alpha)}\}_{\alpha=1}^{p}$ que consta de pares en los cuales $\boldsymbol{\xi}^{(\alpha)}$ es la entrada y $\boldsymbol{\zeta}^{(\alpha)}$ es el resultado deseado asociado a la entrada $\boldsymbol{\xi}^{(\alpha)}$. Para entrenar, se define alguna métrica de distancia o error entre la salida al evaluar $\boldsymbol{\xi}^{(\alpha)}$, y la salida deseada $\boldsymbol{\zeta}^{(\alpha)}$. En cambio, el *aprendizaje no supervisado* usa un conjunto de entrenamiento que no contiene los resultados deseados.

En una red neuronal, el *aprendizaje* consiste en aproximar una función $\boldsymbol{O} = f(\boldsymbol{x})$ de forma que esta sea capaz de calcular $\boldsymbol{O}$ para entradas $\boldsymbol{x}$ no vistas anteriormente. Esto se logra mediante un *algoritmo de entrenamiento*, que suministra a la red entradas del conjunto de entrenamiento y reajusta los pesos de las conexiones entre las neuronas en función de las salidas de la red.

La evaluación del conjunto de entrenamiento completo en el algoritmo de entrenamiento se conoce como época. En principio, se debería entrenar por varias épocas. El problema es que al aumentar el numero de épocas, se corre riesgo de que se produzca un sobreajuste o *overfitting* de la función f. Esto implica que la red predice el conjunto de entrenamiento con una alta precisión, con lo que puede tender a realizar malas predicciones sobre otros conjuntos de datos.

Uno de los algoritmos de entrenamiento más populares es el que combina *backpropagation* con descenso por el gradiente, o alguna de sus variantes. Como se mencionó previamente, se define una función de pérdida $\mathcal{L}(\boldsymbol{\beta})$, o error, y se calcula el gradiente de esta función respecto a los pesos de las conexiones. El algoritmo de backpropagation calcula el gradiente de esta función de pérdida, una capa a la vez, utilizando la regla de la cadena junto con la regla de propagación, iterando hacia atrás desde la capa de salida. Para cada elemento del conjunto de entrenamiento, los pesos se ajustan mediante $\Delta\boldsymbol{\beta} = -\eta\nabla_{\boldsymbol{\beta}}\mathcal{L}(\boldsymbol{\beta})$, de forma que cada ajuste que se haga tienda a minimizar $\mathcal{L}(\boldsymbol{\beta})$ y es por esto que recibe el nombre de descenso por el gradiente. El hiperparámetro η se denomina tasa de aprendizaje, o *learning rate*, y controla la intensidad de los ajustes durante el proceso de aprendizaje.

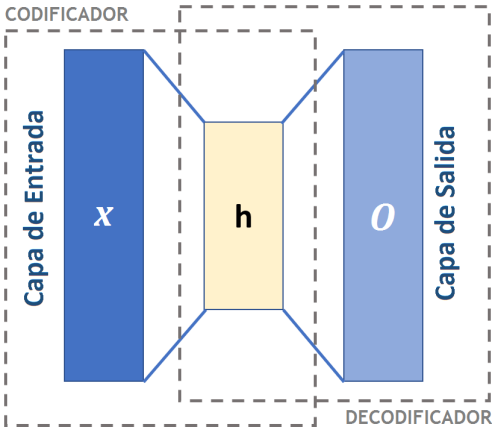

Figura 2: Esquema de un autoencoder de una capa oculta $\boldsymbol{h}$ con entrada $\boldsymbol{x}$ y salida $\boldsymbol{O}$, donde se resaltan las partes que componen el encoder y el decoder. Adaptación[2].

Existen dos grandes inconvenientes con este algoritmo. Primero, la función de pérdida presenta un alto número de puntos críticos, lo que hace que el algoritmo se demore, o quede atrapado, en mínimos locales. Segundo, el algoritmo trata de igual manera a todas las direcciones del espacio sobre el cual $\mathcal{L}(\boldsymbol{\beta})$ esta definida.

III. AUTOENCODERS

Un autoencoder es una red neuronal feedforward totalmente conectada cuyo propósito es que su salida sea igual a su entrada. Típicamente, los autoencoders son simétricos respecto a la capa central y se componen de dos partes. Desde la capa de entrada hasta la capa central, inclusive, se denomina codificador o *encoder*, y desde la capa central, inclusive, hasta la capa de salida, decodificador o *decoder*. Así, se puede dividir el autoencoder en dos redes neuronales, un encoder, que codifica o comprime la entrada a otra representación diferente, y un decoder, que decodifica o descomprime la nueva representación a su formato original.

El autoencoder mas simple tiene una capa oculta $\boldsymbol{h}$, con una menor cantidad de neuronas que las capas de entrada y salida, que describe una codificación usada para representar la entrada. En la Figura 2 se muestra un esquema de un autoencoder de estas características. Se puede pensar que la red aproxima dos funciones: una función encoder $\boldsymbol{h} = \mathcal{E}(\boldsymbol{x})$ y una función decoder $\boldsymbol{O} = \mathcal{D}(\boldsymbol{h})$, donde se busca que su composición $\mathcal{D}(\mathcal{E}(\boldsymbol{x}))$ sea similar a la función identidad y produzca una reconstrucción de la entrada.

Los autoencoders se entrenan de forma que preserven la mayor cantidad de información posible cuando la en-

Figura 3: Imágenes seleccionadas aleatoriamente del dataset Fashion-MNIST.

trada se somete al encoder y al decoder. Se espera que, al entrenar el autoencoder, $\boldsymbol{h}$ aprenda features útiles. Una forma de obtener features útiles del autoencoder es forzar a la capa oculta a una dimensión menor que la dimensión de entrada. Un autoencoder cuya dimensión de codificación es menor que la dimensión de entrada se conoce como *undercomplete*. Aprender una representación undercomplete fuerza al autoencoder a capturar las propiedades salientes del conjunto de entrenamiento. Luego del entrenamiento, y una vez que la representación en una dimensión menor es aprendida, se puede prescindir del decodificador, y el encoder funciona como extractor de características. Es por esto que, tradicionalmente, los autoencoders se usan para feature extraction.

El autoencoder se considera aprendizaje no supervisado ya que no asocia una salida a cada entrada, sino que codifica cada entrada en patrones comunes que surgen en el conjunto de entrenamiento y decodifica las representaciones para volver a obtener la entrada. Al ser redes neuronales feedforward, pueden ser entrenados con las mismas técnicas, típicamente descenso por el gradiente luego de haber computado los gradientes por backpropagation.

IV. AUTOENCODER SOBRE FASHION-MNIST

El conjunto de datos, o *dataset*, utilizado es Fashion-MNIST[3], que esta formado por 70.000 imágenes de 28x28 píxeles en escala de grises, como las que se muestran en la Figura 3, de 10 tipos de prenda de ropa y calzado. El dataset está divido en dos conjuntos, un conjunto de entrenamiento de 60.000 imágenes que es utilizado para entrenar, y un conjunto de prueba de 10.000 imágenes que es utilizado para corroborar el aprendizaje.

Para la implementación se hizo uso de la librería PyTorch[4]. Debido a la naturaleza del dataset, el autoencoder cuenta con $28^2 = 784$ neuronas en las capas de entrada y salida. Se consideró una familia de arquitecturas que varía en el número de neuronas de la capa oculta; 64, 128, 256 y 512 neuronas. Para evaluar que arquitectura ajusta mejor los datos, se usa un conjunto de validación y se toma la arquitectura que de los mejores resultados sobre el conjunto de validación, a través de la función de pérdida o algún otro criterio. En este caso, se utilizó el conjunto de prueba como conjunto de

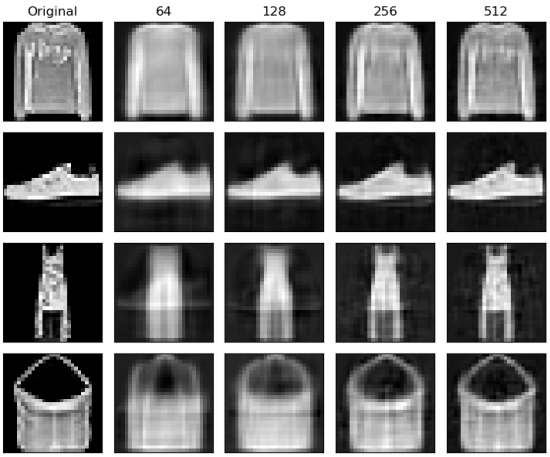

Figura 4: Imágenes seleccionadas aleatoriamente del conjunto de validación junto con las predicciones de cada arquitectura, indicando el correspondiente número de neuronas de la capa oculta.

validación.

La función de activación utilizada fue *Rectified Linear Unit* o ReLU, dada por $\mathrm{ReLU}(z) = \max(0, z)$, y es una función identidad para valores positivos del dominio, y una función nula para valores negativos. Esta función de activación tiene la ventaja de tener un gradiente constante, por lo que su uso no requiere de cálculos computacionalmente costosos. La inicialización de los pesos se hace mediante una distribución uniforme entre $\pm \|\mathcal{N}\|$, donde $\|\mathcal{N}\|$ denota la cantidad de neuronas en la capa anterior.

El error cuadrático medio o MSE (*mean squared error*) entre la entrada y la predicción de la red se utilizó como función de pérdida, tanto para entrenamiento como para validación. Para minimizar los inconvenientes durante el entrenamiento mencionados en la Sección II A, se optó por diversos métodos frecuentemente utilizados.

Como algoritmo de entrenamiento, se utilizó backpropagation junto con una variante de descenso por el gradiente denominada *Adaptive Moment Estimation* o Adam. Adam es un método adaptativo que utiliza el gradiente de la función de pérdida tanto de la época actual como de la época anterior. Al incorporar valores de actualizaciones anteriores y ser computacionalmente eficiente, acelera la convergencia y permite reducir el tiempo de entrenamiento. Se utilizaron los valores predeterminados por PyTorch para los parámetros donde, en particular, la taza de aprendizaje tiene el valor 10^{-3}.

Para evitar overfitting, se utilizó entrenamiento por mini-lotes o *minibatch*, que consiste en dividir el conjunto de entrenamiento en grupos. Se evalúa cada grupo y se calcula los incrementos en los pesos de las conexiones, acumulándolos pero sin actualizarlos. Una vez que se evaluó el grupo completo, se lleva a cabo la actualización de pesos con la suma acumulada. En cada época se hace una resignación aleatoria de todos los elementos del conjunto de entrenamiento. Este método pretende introducir aleatoriedad en el método de descenso por el gradiente al

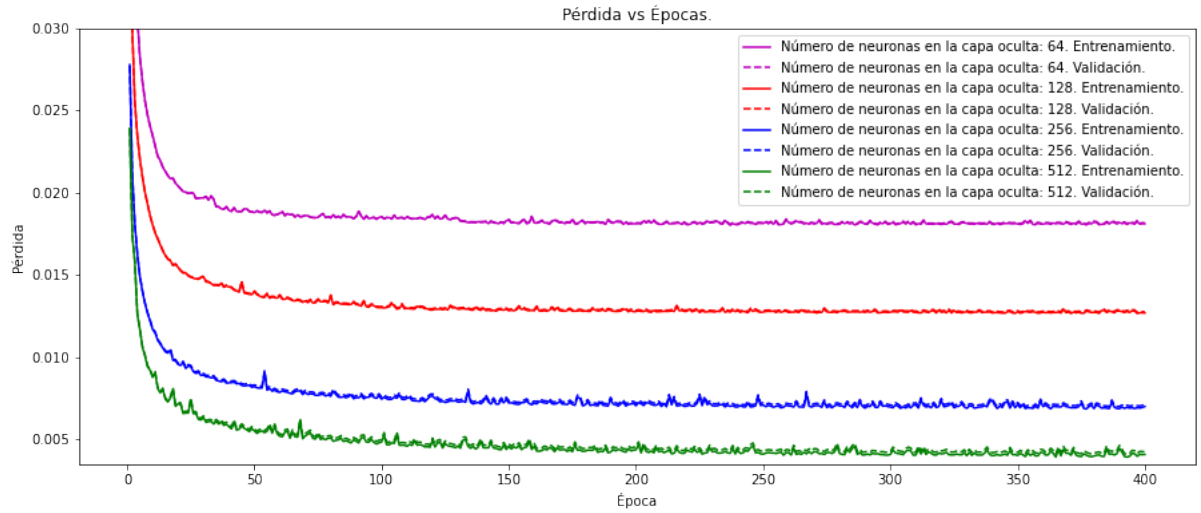

Figura 5: Curva de pérdida en función del numero de épocas, tanto para entrenamiento como para validación.

sortear época a época los elementos del conjunto de entrenamiento, y ayuda con la convergencia. Se utilizaron minibatch de tamaño 1000.

Por estas mismas razones se utilizó *dropout*, que consiste en suprimir temporalmente neuronas, de forma aleatoria con alguna probabilidad, para cada entrada durante la fase de entrenamiento. Durante la fase de backpropagation, estas neuronas tampoco están activas, por lo que los pesos de sus conexiones no se actualizan. Con esto se pretende reducir correlaciones entre neuronas, lo que evita overfitting, y evitar la convergencia a mínimos locales al introducir ruido. Se utilizó dropout con una probabilidad de 0,1.

Cada arquitectura se entrenó por 400 épocas. En la Figura 4 se muestran algunas predicciones, mientras que en la Figura 5 se puede observar la pérdida en cada época en función del numero de épocas. En un principio se tomó 400 como un número grande arbitrario, ya que se esperaba que el error de validación comience a aumentar en algún momento del entrenamiento, lo que indica signos de overfitting. Como se puede observar, esto no sucedió y es por esto que se conservó ese numero de épocas.

V. CONCLUSIÓN

El autoencoder de una capa oculta es de los algoritmos de representation learning más simples. Tiene una arquitectura sencilla y es muy útil para ilustrar un gran abanico de conceptos.

La implementación funciona de acuerdo a lo esperado. Se puede observar claramente en la Figura 4 como mejoran las predicciones cuando crece el número de neuronas de la capa oculta y el autoencoder es capaz de captar más features. Es fácil ver también que las predicciones del autoencoder no son del todo precisas con un bajo número de neuronas pero adquieren mas detalle cuando este número aumenta.

Las curvas de entrenamiento y validación mostradas en la Figura 5 son similares, para las cuatro arquitecturas, lo que sugiere ausencia de overfitting, y presentan una tendencia general al aplanamiento, lo que indica convergencia.

[1] Adaptación (CC BY 4.0). Obra original por W.-J. Niu, Z.-K. Feng, B.-F. Feng, Y.-W. Min, C.-T. Cheng & J.-Z. Zhou. "*Comparison of Multiple Linear Regression, Artificial Neural Network, Extreme Learning Machine, and Support Vector Machine in Deriving Operation Rule of Hydropower Reservoir*". Water. 2019; 11(1):88. `https://doi.org/10.3390/w11010088`

[2] Adaptación (CC BY 4.0). Obra original por G. E. Hinton & R. R. Salakhutdinov. "*Reducing the Dimensionality of Data with Neural Networks*". Science. 2006; 313(5786):504. `https://doi.org/10.1126/science.1127647`

[3] H. Xiao, K. Rasul & R. Vollgraf. "*Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*". arXiv. Aug 2017. `https://doi.org/10.48550/arxiv.1708.07747`. Dataset disponible en: `https://github.com/zalandoresearch/fashion-mnist`

[4] A. Paszke et al. "*PyTorch: An Imperative Style, High-Performance Deep Learning Library*". Advances in Neural Information Processing Systems 32. 2019; pp. 8024–8035. Disponible en: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.