Autoencoder Convolucional para Clasificación en FashionMNIST

Vocos, Augusto*

Facultad de Ciencias Económicas, Universidad Nacional de Córdoba

Facultad de Astronomía, Matemática y Física, Universidad Nacional de Córdoba

28 de Febrero de 2025

Abstract

En este trabajo se describe el diseño, implementación y evaluación de un autoencoder convolucional aplicado a la base de datos *FashionMNIST*, para después ser utilizado en la clasificación. Se detalla la arquitectura del modelo, y se discuten tanto la calidad de la reconstrucción como el desempeño en tareas de clasificación.

1 Introducción

El presente trabajo tiene como objetivo desarrollar y optimizar un autoencoder convolucional, inicialmente para la reconstrucción de imágenes y posteriormente para su clasificación, utilizando la libería PyTorch[1] de Python. Se explorarán distintas configuraciones de hiperparámetros y arquitecturas de red con el fin de determinar la mejor red posible.

Para entrenar y evaluar el modelo, se utilizó la base de datos ***FashionMNIST*** [2], un conjunto de 70,000 imágenes en escala de grises de 28×28 píxeles, distribuidas en 10 clases de artículos de ropa y accesorios.

2 Arquitectura del Autoencoder Base

La arquitectura se divide en dos componentes principales: un módulo de codificación (*encoder*) y otro de decodificación (*decoder*). El tamaño de las salidas (para las capas que modifican el input, es decir, las convoluciones y el Max Pooling) puede calcularse de la siguiente manera:

$$O = \frac{(I - K + 2P)}{S} + 1$$

Donde: - O es el tamaño de la salida (output). - I es el tamaño de la entrada (input). - K es el tamaño del kernel. - P es el padding. - S es el stride.

2.1 Encoder

- **Capa convolucional 1**: 16 kernels de 3×3, activación ReLU, y *dropout* ($p = 0.2$). Transforma la entrada $(1, 28, 28)$ en una salida de $(16, 26, 26)$.
- **Max Pooling**: Kernel de 2×2 con stride 2, reduciendo el input a $(16, 13, 13)$.
- **Capa convolucional 2**: 32 kernels de 3×3, ReLU, y *dropout*. Produce un output de $(32, 11, 11)$.
- **Max Pooling**: Reduce el input a $(32, 5, 5)$.
- **Capa Flatten**: Convierte el input en un vector unidimensional de tamaño $32 \times 5 \times 5 = 800$.
- **Capa lineal**: Proyecta el vector a una representación latente de tamaño n (hiperparámetro ajustable), con ReLU y *dropout*.

*augusvocos@mi.unc.edu.ar

2.2 Decoder

El *decoder* reconstruye la imagen original desde la representación latente:

- **Capa lineal**: Transforma la representación latente (de tamaño n) a 32×5×5, con ReLU y *dropout*.
- **Capa Unflatten**: Transforma el vector en un mapa de características de (32,5,5).
- **Capa transpuesta convolucional 1**: 16 filtros de 5×5, *stride* 2, y parámetros de padding ajustados. Genera un mapa de (16,13,13) con ReLU y *dropout*.
- **Capa transpuesta convolucional 2**: 1 filtro de 6×6, *stride* 2, y activación sigmoide. Recupera la resolución original de (1,28,28).

3 Metodología

Para la selección del mejor autoencoder convolucional, se implementó una metodología basada en la exploración sistemática de hiperparámetros mediante un *árbol de decisión*. Si bien se evaluaron distintas configuraciones del modelo variando parámetros clave, la secuencialidad de la elección implica que, en una primera instancia, fue necesario elegir un conjunto base de hiperparámetros. Se utilizó una probabilidad de dropout de 20%($p = 0.2$), tasa de aprendizaje de 0.001 (o learning rate, $lr = 0.001$), un tamaño de bache de 100 ($bs = 100$), el optimizador $ADAM$ y la arquitectura convolucional descripta en la sección anterior. Las comparaciones de desempeño se hizo en base al cálculo del error cuadrático medio (ECM) sobre los conjuntos de entrenamiento y validación.

4 Búsqueda de hiperparámetros

En primer lugar, se experimentó con diferentes dimensiones de la capa lineal final del encoder (y, consecuentemente, con la primera capa del decoder), considerando valores de $n = 64, 128, 256$. A partir de los resultados obtenidos en la Figura 1 (a), se seleccionó $n = 256$, dado que proporcionó una mejor representación de los datos sin inducir sobreajuste. Se destaca que el modelo con $n = 64$ no solo muestra un error de ajuste mayor, sino que denota una variabilidad mucho más alta que las otras configuraciones.

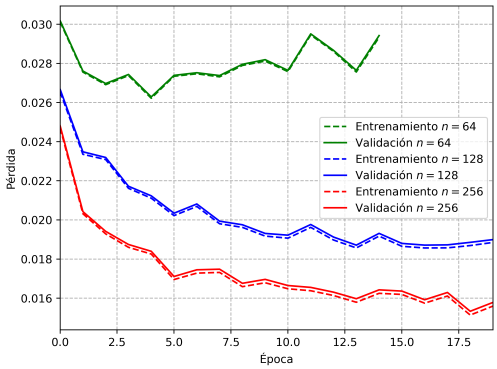
Posteriormente, se ajustó el factor de *dropout* p, comparando $p = 0.2$ y $p = 0.1$ y seleccionando el valor menor debido a su impacto positivo en la regularización del modelo. Véase Figura 1 (b).

Seguidamente se analizaron distintos valores de la tasa de aprendizaje (lr), evaluando $lr = 0.01$ y $lr = 0.001$. La configuración con $lr = 0.001$ mostró una convergencia más estable y una menor variabilidad en la pérdida. De manera similar, se exploró la influencia del tamaño del lote (bs), probando $bs = 200, 100, 50$; y se determinó que $bs = 50$ resultó en un mejor rendimiento. Los resultados pueden observarse en las Figuras 1 (c) y (d), respectivamente.

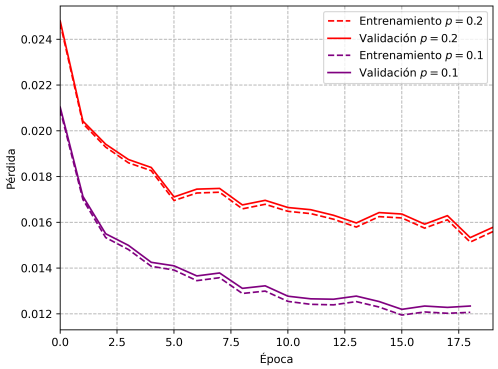
Finalmente, se compararon dos algoritmos de optimización: **SGD** y **ADAM**, donde **ADAM** se seleccionó por su capacidad para acelerar la convergencia y mejorar la reducción del error de reconstrucción del autoencoder: los resultados pueden verse en la Figura 1 (e). Asimismo, se evaluaron dos arquitecturas distintas:

- **Primera arquitectura: base**[1]
- **Segunda arquitectura: alternativa**
  - Capa Convolucional 1 de (1, 28, 28) → (32, 28, 28)
  - Max Pooling de (32, 28, 28) → (32, 14, 14)
  - Capa Convolucional 2 de (32, 14, 14) → (64, 14, 14)
  - Max Pooling de (64, 14, 14) → (64, 7, 7)
  - Capa Flatten de (64, 7, 7) → 3136
  - Capa lineal de 3136 a 256

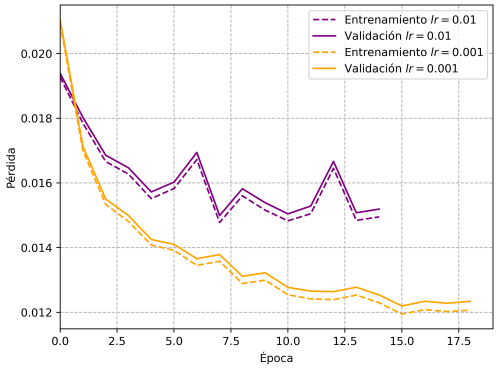
[1] Descripta en sección 2 de este trabajo.



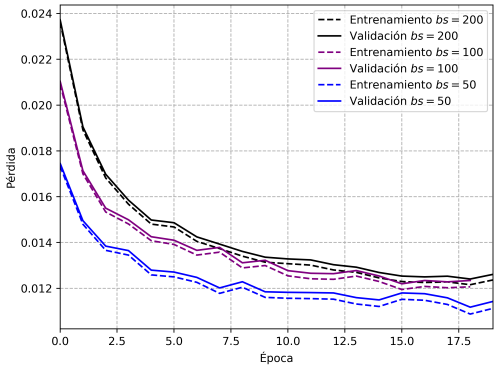
(a) Modificando capa lineal n



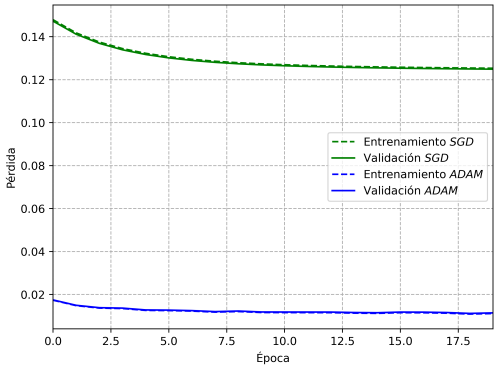
(b) Modificando probabilidad de dropout p



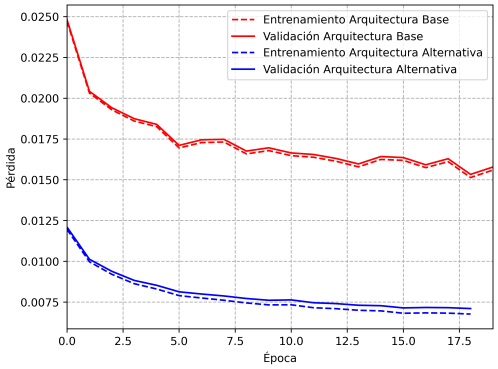
(c) Modificando tasa de aprendizaje lr



(d) Modificando tamaño de bache bs



(e) Modificando optimizador

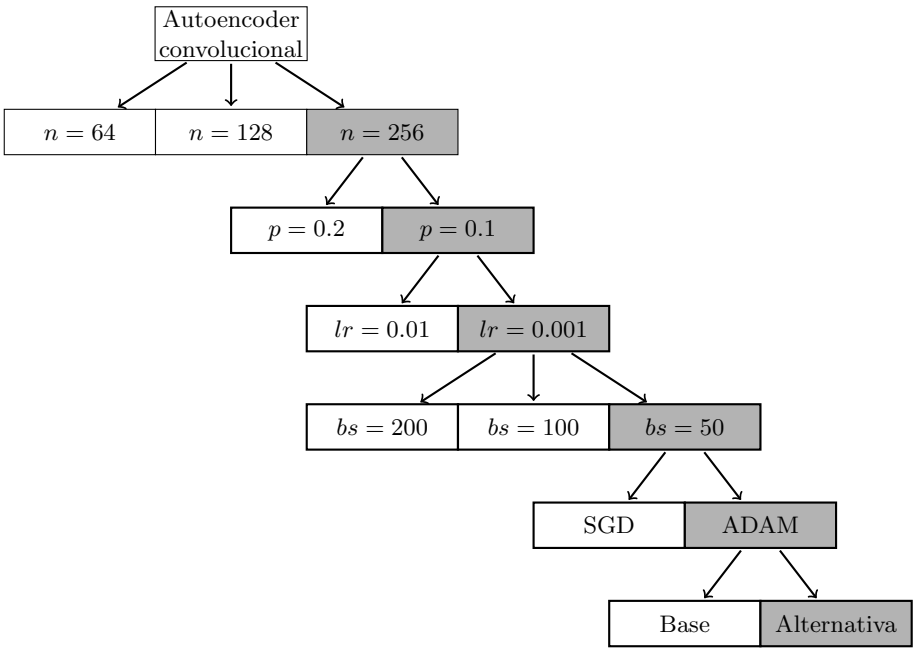


(f) Modificando arquitectura

Figura 1: Variaciones en hiperparámetros n, p, lr, bs, optimizador, arquitectura

Los resultados pueden apreciarse en la Figura 1 (f). La segunda arquitectura se seleccionó como la mejor opción, ya que permitió una mejor representación de los datos con menor pérdida de información y mejor desempeño en validación.

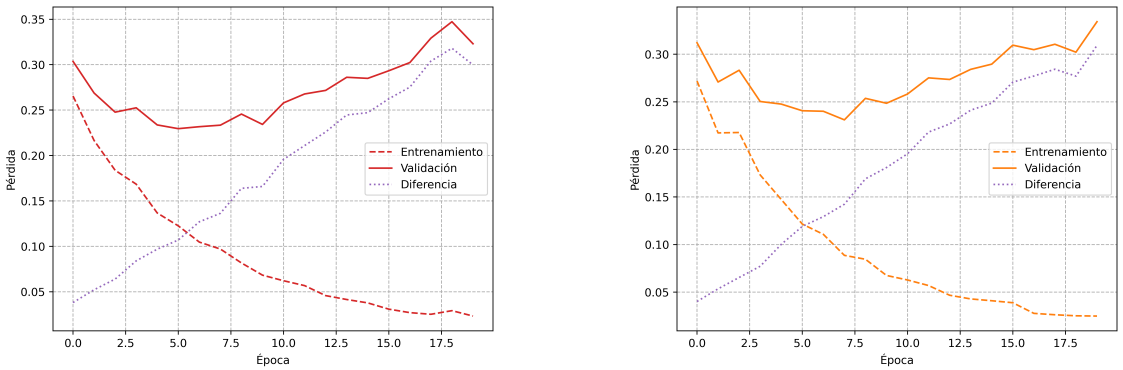
En resumen, el mejor autoencoder convolucional encontrado utilizó los siguientes hiperparámetros: $n = 256$, $p = 0.1$, $lr = 0.001$, $bs = 50$, el optimizador **ADAM** y la arquitectura convolucional alternativa, descrita anteriormente. A modo de ilustración, a continuación se encuentra un esquema del árbol secuencial de decisión resultante:



5 Entrenado el Clasificador

Una vez que se ha entrenado y optimizado el autoencoder convolucional, el siguiente paso es construir y entrenar un clasificador. El objetivo es evaluar los efectos del entrenamiento del encoder y el clasificador, tanto en conjunto como por separado, sobre la clasificación de la base de datos *FashionMNIST*.

El clasificador consiste en una red neuronal simple que toma como entrada la representación latente de tamaño $n = 256$ y la transforma en una salida de 10 clases, correspondientes a las categorías de *FashionMNIST*. Cabe destacar que, para el entrenamiento del problema de clasificación, se utilizó la función de pérdida *Cross Entropy Loss*.



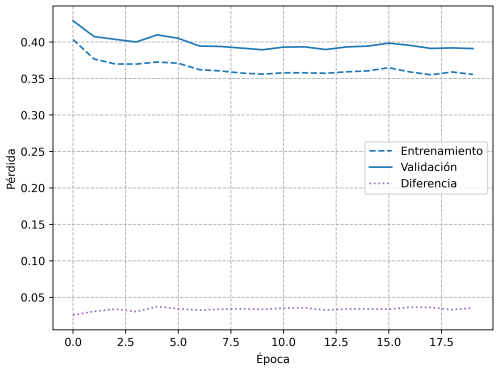
(a) Encoder entrenado

(b) Encoder sin entrenar

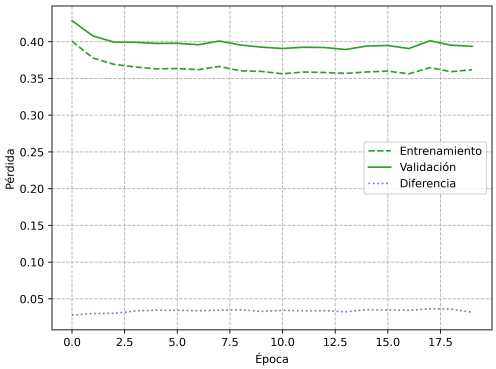
Figura 2: Error; entrenando Encoder y Clasificador

La Figuras 2 y 3 muestran el comportamiento de los errores de entrenamiento y validación, dependiendo qué módulo/s se elige entrenar. La estabilidad de la curva **Diferencia**, de color violeta, refleja la ausencia de sobreajuste del modelo, pues errores de entrenamiento y validación progresan de manera similar a lo largo de las épocas. De la misma manera, la pendiente positiva de la misma evidencia una separación entre los errores; mientras el error de entrenamiento decrece (es decir, la red aprende a clasificar el conjunto de entrenamiento), la generalización de este comportamiento a ejemplos no vistos no progresa de la misma manera. Entonces, se confirma la presencia de sobreajuste del modelo.

Como primera conclusión, es posible observar que los modelos presentan sobreajuste cuando se entrena el autoencoder en conjunto con el clasificador; así lo muestran las Figuras 2 (a) y (b). Por otro lado, las Figuras 3 (a) y (b) permiten afirmar que entrenar el clasificador por separado, sea con un encoder entrenado o no, no crea sobreajuste del modelo.



(a) Encoder entrenado

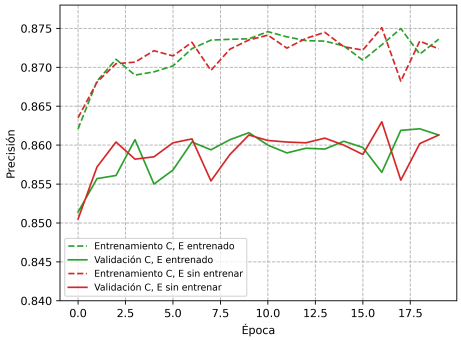


(b) Encoder sin entrenar

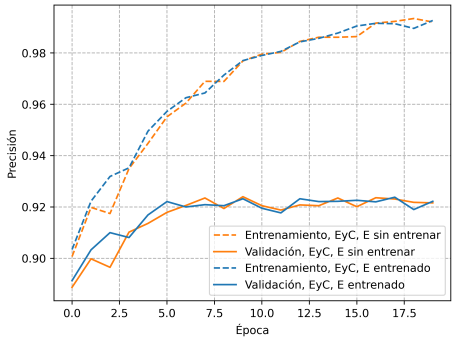
Figura 3: Error; entrenando Clasificador

Por otra parte, el valor absoluto de los errores de predicción es menor para el caso en que encoder y clasificador se entrenan de manera conjunta. Si se evalúa junto con el punto anterior, parecería haber un *trade-off* entre generalización y precisión, basada en la elección de entrenar el encoder y el clasificador de manera conjunta o separada.

Por último, a partir de las Figuras 3 (a) y (b) cabe destacar que, además de reflejar una performance muy similar, los modelos no cuentan con una capacidad notable de reducción del error, independientemente de si el autoencoder ha sido entrenado previamente o no. De esta manera, se extraen dos conclusiones: en primer lugar, la performance del clasificador es independiente del entrenamiento previo del autoencoder; por otro lado, entrenar el clasificador de manera separada del encoder no parece tener efectos en la reducción del error de predicción.



(a) Entrenando Clasificador



(b) Entrenando Encoder y Clasificador

Figura 4: Precisión

La Figura 4 muestra la precisión de predicción de los modelos. Han sido agrupados en base a las similitudes analizadas en los errores: la Figura 4 (a) expone la precisión alcanzada en el entrenamiento del Clasificador por separado, utilizando tanto el encoder entrenado como sin entrenar. Por su parte, la Figura 4 (b) muestra la performace predictiva obtenida con el entrenamiento del encoder y clasificador de manera conjunta, donde también se hizo distinción entre un encoder inicial sin entrenar y uno entrenado.

Como puede observarse, si bien existe sobreajuste, la capacidad predictiva del modelo es superadora para el caso en que encoder y clasificador se entrenan de manera simultánea. De esta manera, los resultados obtenidos en el análisis del comportamiento del error se extienden a la precisión del modelo.

Por último, en la Figura 5 se presentan la matriz de confusión para el caso expuesto en la Figura 2 (b), a saber: entrenando encoder y clasificado de manera conjunta, a partir de un encoder sin entrenar. Una matriz de confusión indica con qué frecuencia las predicciones de un modelo coinciden con las etiquetas reales, proporcionando una herramienta clave para analizar su rendimiento en tareas de clasificación. Estos cuadros permiten visualizar errores y aciertos en las categorías asignadas.

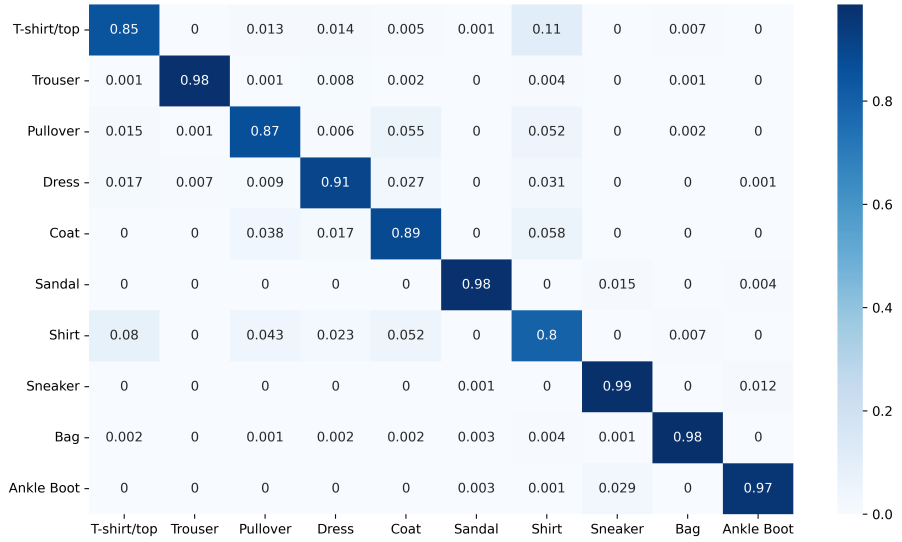


Figura 5: Matriz de confusión, Encoder y Clasificador entrenados de manera conjunta, Encoder sin entrenar

6 Conclusiones

Este estudio implementó un autoencoder convolucional para reconstrucción y clasificación en el conjunto *FashionMNIST*, utilizando una metodología secuencial de selección de hiperparámetros basada en un árbol de decisiones estructurado. Si bien este enfoque permitió identificar una configuración óptima, su naturaleza secuencial podría restringir la exploración integral de combinaciones de parámetros. Para abordar esta limitación, futuros trabajos podrían emplear técnicas alternativas, que optimizarían el análisis del espacio de hiperparámetros.

Los resultados mostraron que el entrenamiento conjunto del *encoder* y el clasificador incrementó la precisión predictiva, pero a expensas de un mayor sobreajuste. En contraste, entrenar ambos módulos por separado redujo este riesgo, aunque con un rendimiento inferior. Curiosamente, la representación latente generada por el *encoder* preentrenado no mejoró significativamente la clasificación, lo que sugiere un clasificador convolucional podría no beneficiarse plenamente de este enfoque.

Para avanzar, se propone explorar otras arquitecturas que potencien la capacidad de representación del modelo. Paralelamente, la aplicación técnicas de regularización avanzadas podría perfeccionar el equilibrio entre precisión y generalización, abriendo nuevas vías para aplicaciones en visión artificial con conjuntos de datos complejos.

References

PyTorch Team. *PyTorch: An Open Source Machine Learning Framework*. 2024. Disponible en: `https://pytorch.org`.

Zalando Research. *Fashion-MNIST dataset*. 2024. Disponible en: `https://github.com/zalandoresearch/fashion-mnist`.