Redes Neuronales
Reconstrucción y Clasificación de Imágenes

Tiago Bruno, *estudiante de la Facultad de Ciencias Económicas de la UNC*

***Resumen*—Este trabajo implementa *autoencoders* y clasificadores sobre *Fashion-MNIST*. Se comparan distintos tipos y arquitecturas de redes con el objetivo de reflexionar sobre la elección de hiperparámetros. Se concluyó que elegir los hiperparámetros es una cuestión de carácter teórico-práctico que depende de la naturaleza del problema que se trate y del tipo de redes implementadas. Para este conjunto de datos en particular, las redes convolucionales son superiores a las *feed-forward*, tanto para comprimir y reconstruir imágenes como para clasificarlas.**

I. Introducción

Este trabajo implementa *autoencoders* y clasificadores sobre *Fashion-MNIST*. Presenta diferentes aplicaciones de los problemas de reconstrucción y clasificación de imágenes, con la finalidad de explorar el proceso de elección de hiperparámetros a la hora de entrenar los modelos. Se utilizan redes *feed-forward* y convolucionales, con distintas arquitecturas. El algoritmo de optimización utilizado es el ADAM (*Adaptative Moment Estimation*), una versión mejorada del descenso por el gradiente estocástico (SGD).

La primera sección comenta brevemente el contenido teórico para comprender cómo funcionan los *autoencoders* y clasificadores convolucionales. Además, presenta las redes implementadas.

La sección siguiente explica, con un enfoque teórico, los problemas que surgen con el método del descenso por el gradiente y sus relaciones con la elección de hiperparámetros.

El libro de Goodfellow (2016) [1] abarca de manera precisa el funcionamiento de las neuronas convolucionales, los problemas del descenso por el gradiente y sus relaciones con los hiperparámetros.

En tercer lugar, se exponen los rendimientos alcanzados por los modelos y se explica el proceso práctico de elección de hiperparámetros. Además, se aplica una técnica de *transfer learning* para los clasificadores, a partir de los *encoders*. En todas las aplicaciones se utilizó el optimizador ADAM.

Finalmente, el trabajo concluye que elegir los hiperparámetros es una cuestión de carácter teórico-práctico que depende de la naturaleza del problema que se trate y el tipo de redes implementadas. Para este conjunto de datos en particular, las redes convolucionales son superiores a las redes *feed-forward*, tanto para comprimir y reconstruir imágenes como para clasificarlas. Se sugiere el estudio de técnicas para elegir los hiperparámetros.

II. Arquitecturas

¿Qué es un *autoencoder* convolucional?

El *autoencoder* es un algoritmo de aprendizaje que aplica *back-propagation* y permite el preprocesamiento de los datos con el objetivo de que la salida sea igual a la entrada. La peculiaridad de este algoritmo consiste en fijar un número de neuronas en la capa intermedia que sea inferior a la cantidad de neuronas en las capas de entrada y salida. La limitación del número de neuronas en la capa oculta obliga a la red a encontrar patrones entre los elementos, lo que permite aprender una representación comprimida de la entrada y reducir la dimensión inicial de los datos. En este sentido, partir de redes que han sido autoaprendidas con el *autoencoder* puede lograr que el método de *back propagation* resulte más eficiente.

La convolución es una operación matemática que se utiliza para procesar datos de manera local. En el contexto de las redes neuronales convolucionales, la convolución implica la aplicación de un filtro (o *kernel*) a una región específica de los datos de entrada.

El filtro se desliza a través de la entrada, multiplica los valores de los píxeles por los valores del filtro y los suma para producir un único valor de salida. El *stride* es la cantidad de píxeles que el filtro se desliza en cada paso durante la convolución. Por lo tanto, cada neurona convolucional está conectada solo a un pequeño parche de la entrada en lugar de a toda la entrada, lo que permite que la red aprenda patrones locales. Así, estas neuronas comprimen los datos, es decir, reducen la dimensión espacial de los datos de entrada.

En estas redes, los datos de entrada pueden tener múltiples canales. Cada canal representa una característica específica de los datos de entrada. Por ejemplo, una imagen en color RGB tiene tres canales: rojo, verde y azul. La base *Fashion-MNIST* está en escala de grises por lo que tiene un canal. Generalmente, las capas convolucionales están seguidas por funciones de *pooling* y capas totalmente conectadas (*fully conected*).

Por otro lado, la convolución transpuesta es la operación inversa de la convolución. Los filtros en la convolución transpuesta aumentan las dimensiones mediante la creación de nuevos valores en función de los datos de entrada.

En resumen, las neuronas convolucionales extraen características locales de los datos de entrada mediante la aplicación de convoluciones, mientras que las convoluciones transpuestas realizan la operación inversa y aumentan la dimensión de los datos, por lo que suelen utilizarse en tareas de reconstrucción, como los *autoencoders*.

La figura 1 presenta un esquema de un *autoencoder* convolucional para ilustrar este tipo de algoritmos. A continuación, se exponen las arquitecturas implementadas en este trabajo.

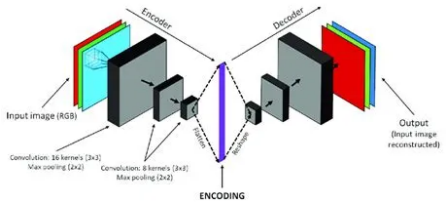

Figura 1. Ilustración de un *autoencoder* convolucional, recuperada de [2]

Autoencoders implementados

Este trabajo implementa tres *autoencoders*, uno *feed-forward* (o lineal) y dos convolucionales, de los cuales a uno lo llamaremos «consigna» y a otro, «Raschka» porque está inspirado en el trabajo de Sebastián Raschka [3].

En la figura 2, se expone el código del *autoencoder feed-forward*, que está compuesto por una capa de entrada de 784, una capa oculta

de 512 y otra de salida también de 784 neuronas. Este tamaño de la capa oculta es el que mejor reconstruye las imágenes, aunque no logre comprimir demasiado los datos. La capa oculta tiene una función de activación *ReLU* mientras que la de salida usa una sigmoide, que genera salidas en el rango $[0, 1]$, lo que es apropiado para el caso de imágenes en escala de grises.

```python
# 2. Autoencoder
class Autoencoder(nn.Module):
    def __init__(self, hidden_size=512, dropout_rate=0.1):
        super().__init__()
        self.encoder = nn.Linear(28*28, hidden_size)
        self.dropout = nn.Dropout(p=dropout_rate)
        self.decoder = nn.Linear(hidden_size, 28*28)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x.view(x.size(0), -1) # vector 1x784
        x = torch.relu(self.encoder(x))
        x = self.dropout(x)
        x = self.sigmoid(self.decoder(x))
        return x
```

Figura 2. *Autoencoder feed-forward* con 512 neuronas en la capa oculta

Antes de explicar las estructuras de los *autoencoders* convolucionales, es conveniente mostrar fórmulas de cálculo para las dimensiones de salida. Las neuronas convolucioanles reducen la dimensionalidad de la siguiente forma:

$$D_s = pe[\frac{D_e - K + 2P}{S}] + 1 \tag{1}$$

donde D_s es la dimensión de salida, D_e la de entrada, K el tamaño del *kernel*, P el *padding* (relleno), S el *stride* y $pe[x]$ la función parte entera suelo.

Por otro lado, las convoluciones transpuestas amplían la dimensión de entrada de esta manera:

$$D_s = (D_e - 1)S - 2P + K + P_s \tag{2}$$

donde P_s es el *output padding*.

Adicionalmente, el *autoencoder* convolucional «consigna» aplica dos funciones de *pooling*. Siguiendo a Goodfellow (2016) [1], una función de *pooling* reemplaza la salida de la red en un lugar determinado con una estadística resumida de las salidas cercanas. Por ejemplo, si se usa max la operación informa el mayor valor dentro de un rectángulo vecindario. Estas funciones reducen la dimensionalidad de la misma forma que las neuronas convolucionales.

La figura 3 presenta la arquitectura del *autoencoder* convolucional «consigna», a modo de ayuda está comentado el cálculo de las dimensiones de salida en cada capa. El *encoder* está compuesto por una capa convolucional que mapea dimensiones $(1, 28, 28)$ a $(16, 26, 26)$, un *maxpooling* que devuelve $(16, 13, 13)$, otra capa convolucional con salida $(32, 11, 11)$, otro *maxpooling* que lleva a $(32, 5, 5)$, una función *flatten* que aplana su entrada a un vector de orden $(32 * 5 * 5)$ y una capa *fully conected* que devuelve un vector del orden del tamaño de la capa oculta (*hidden size*). El *decoder* está formado por una capa lineal que expande el vector de la capa oculta a otro de orden $(32 * 5 * 5)$, una capa *unflatten* que arma tensores $(32, 5, 5)$, una capa convolucional transpuesta que devuelve $(16, 13, 13)$ y otra capa convolucional transpuesta que reconstruye $(1, 28, 28)$. Entre capa y capa hay *dropout* y se utiliza la función de activación *ReLU*, salvo en la salida que usa una *sigmoid*.

```python
class Autoencoder(nn.Module):
    def __init__(self, hidden_size=512, dropout_rate=0.1):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3), # Conv1    28-3+1=26 (16,26,26)
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.MaxPool2d(kernel_size=2), # MaxPool1 pe[(26-2)/2]+1=13 (16,13,13)
            nn.Conv2d(16, 32, kernel_size=3), # Conv2    13-3+1=11 (32,11,11)
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.MaxPool2d(kernel_size=2), # MaxPool2 pe[(11-2)/2]+1=5 (32,5,5)
            nn.Flatten(),
            nn.Linear(32 * 5 * 5, hidden_size), # Fully Conected
            nn.ReLU(),
            nn.Dropout(dropout_rate)
        )
        self.decoder = nn.Sequential(
            nn.Linear(hidden_size, 32 * 5 * 5),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Unflatten(1, (32, 5, 5)),
            nn.ConvTranspose2d(32, 16, kernel_size=4, stride=2, output_padding=1),
            # ConvTransp1 (5-1)*2+4+1=13 (16,13,13)
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.ConvTranspose2d(16, 1, kernel_size=3, stride=2, output_padding=1),
            # ConvTransp2 (13-1)*2+3+1=28 (1,28,28)
            nn.Sigmoid(),
            nn.Dropout(dropout_rate)
```

Figura 3. *Autoencoder* convolucional «consigna»

En la figura 4 se expone la arquitectura del *Autoencoder* convolucional «Raschka». El *encoder* tiene cuatro capas convolucionales, una función *flatten* y una capa lineal que comprime la entrada al tamaño de la capa oculta, mientras el *decoder* cuenta con una capa lineal, una función *unflatten* y cuatro capas convolucionales transpuestas. Al final, usa una función llamada *Trim()* que se encarga de que la salida tenga las mismas dimensiones que la entrada. Tiene *dropout* entre las capas y usa la función de activación *Leaky ReLU*, salvo en la salida que usa una *sigmoid*.

```python
self.encoder = nn.Sequential( #784
        nn.Conv2d(1, 32, stride=(1, 1), kernel_size=(3, 3), padding=1), # 1 28-3+2+1=28 (32,28,28)
        nn.LeakyReLU(0.01),
        nn.Dropout(dropout_rate),
        nn.Conv2d(32, 64, stride=(2, 2), kernel_size=(3, 3), padding=1), # 2 pe[(28-3+2)/2]+1=14 (64,14,14)
        nn.LeakyReLU(0.01),
        nn.Dropout(dropout_rate),
        nn.Conv2d(64, 64, stride=(2, 2), kernel_size=(3, 3), padding=1), # 3 pe[(14-3+2)/2]+1=7 (64,7,7)
        nn.LeakyReLU(0.01),
        nn.Dropout(dropout_rate),
        nn.Conv2d(64, 64, stride=(1, 1), kernel_size=(3, 3), padding=1), # 4 7-3+2+1=7 (64,7,7)
        nn.Flatten(),
        nn.Linear(64 * 7 * 7, hidden_size)                               # fc
)
self.decoder = nn.Sequential(
        nn.Linear(hidden_size, 64 * 7 * 7),
        nn.Unflatten(1, (64, 7, 7)),
        nn.ConvTranspose2d(64, 64, stride=(1, 1), kernel_size=(3, 3), padding=1), # 1 (7-1)-2+3=7 (64,7,7)
        nn.LeakyReLU(0.01),
        nn.Dropout(dropout_rate),
        nn.ConvTranspose2d(64, 64, stride=(2, 2), kernel_size=(3, 3), padding=1), # 2 (7-1)*2-2+3=13 (64,13,13)
        nn.LeakyReLU(0.01),
        nn.Dropout(dropout_rate),
        nn.ConvTranspose2d(64, 32, stride=(2, 2), kernel_size=(3, 3), padding=0), # 3  (13-1)*2+3=27 (32,27,27)
        nn.LeakyReLU(0.01),
        nn.Dropout(dropout_rate),
        nn.ConvTranspose2d(32, 1, stride=(1, 1), kernel_size=(3, 3), padding=0),  # 4  (27-1)+3=29 (1,29,29)
        Trim(),  # 1x29x29 -> 1x28x28
        nn.Sigmoid()
```

Figura 4. *Autoencoder* convolucional «Raschka»

Clasificadores

Este trabajo implementa dos clasificadores, uno parte del *encoder* de «consigna» y el otro del «Raschka». Se modificaron los *dropouts* de los *encoders*. En el caso del «consigna», se eliminaron los dos primeros y el tercero se movió luego de la operación *flatten*. Respecto al «Raschka», se eliminaron todos los *dropout* y se agregó uno luego de *flatten*.

La capa de clasificación tiene una entrada del tamaño de la capa oculta y devuelve un vector de orden 10, ya que la base de datos tiene diez clases. No se utiliza ninguna función de activación en la capa de salida, por lo que los vectores de salida contendrán escalares de cualquier valor. Para obtener la clasifiación se elige el orden del mayor elemento del vector.

Otra forma de implementar clasificadores pordría ser usar una función de activación *softmax* en la capa de salida, para que los

elementos del vector estén en el rango $[0, 1]$ y sumados den uno. Así, cada elemento del vector representa la confianza del clasificador en la pertenencia del objeto de entrada a la clase etiquetada con su correspondiente orden. La clase con la «probabilidad» más alta se considera la predicción del clasificador. Este enfoque fue aplicado, no obstante, los modelos alcanzaban rendimientos inferiores a los del enfoque explicado anteriormente, que es llamado *argmax*.

En este trabajo, para evaluar el desempeño de los clasificadores se utiliza la *Cross Entropy Loss* (CEL) como función de pérdida a optimizar, junto con la Precisión (*Accuracy*).

La *Cross Entropy Loss* es una medida de la diferencia entre dos distribuciones de probabilidad: la distribución de probabilidad predicha por el modelo y la distribución de probabilidad real de los datos. Esta función de pérdida se utiliza para ajustar los pesos del modelo durante el entrenamiento.

La precisión es la cantidad de predicciones correctas realizadas por el modelo sobre el total de predicciones realizadas.

En resumen, la *Cross Entropy Loss* mide qué tan bien el modelo aprende los datos y ajusta los pesos, mientras que la precisión evalúa qué tan bien el modelo clasifica los datos. Ambas métricas son fundamentales para evaluar el rendimiento de los clasificadores.

III. La Elección de Hiperparámetros

Los hiperparámetros son parámetros que no se aprenden directamente del conjunto de datos durante el proceso de entrenamiento de un modelo, sino que se establecen anteriormente. Estos objetos controlan aspectos del proceso de entrenamiento y afectan la capacidad y la velocidad de aprendizaje del modelo.

Estos hiperparámetros son configurables y deben ser ajustados de forma manual o mediante técnicas de optimización, para lograr un rendimiento óptimo del modelo en una tarea específica. Este trabajo se concentra principalmente en el número de épocas, el tamaño de la capa oculta, el tamaño de lotes (*mini-batchs*), la tasa de aprendizaje y el *dropout*. Además, reflexiona brevemente sobre elecciones relacionadas a las arquitecturas, particularmente sobre las funciones de activación, las funciones de *pooling* y la cantidad de conexiones sinápticas.

El método del descenso por el gradiente (y sus variaciones como el SGD o el ADAM) tiene obstáculos para su implementación, este trabajo se enfoca en tres de ellos: la dependencia de valores iniciales de los pesos, el estancamiento del gradiente y el sobreajuste. Una elección adecuada de los hiperparámetros contribuye a solucionar estos problemas. Es importante tener en cuenta que hay relaciones entre estos objetos, por lo que no se trata de elegirlos por separado, sino por encontrar una buena combinación.

En primer lugar, la dependencia de valores iniciales de los pesos surge porque la evolución del error está relacionada a estos valores que se eligen inicialmente al azar, por lo que alcanzar un mínimo local dependerá de la probabilidad de «caer» cerca de ellos al seleccionar los parámetros. Una forma de superar esta cuestión es el *transfer learning*, esta técnica busca aprovechar el conocimiento adquirido por un modelo entrenado en una tarea y aplicarlo a otra tarea relacionada.

Dos maneras de aplicar el *transfer learning* son:

- *Fine-tuning* (ajuste fino): se toma un modelo pre-entrenado y se ajustan los pesos durante el entrenamiento del nuevo modelo, con diferentes elecciones de hiperparámetros.
- *Feature extraction* (extracción de características): se toma un modelo pre-entrenado y se utilizan sus capas internas para extraer características de los datos de entrada, que se utilizan para iniciar un nuevo modelo. Esto permite aprovechar el conocimiento previamente aprendido por el modelo pre-entrenado y aplicarlo a nuevas tareas o datos.

En segundo lugar, el estancamiento del gradiente ocurre cuando los gradientes calculados durante el *backpropagation* son demasiado pequeños, lo que dificulta el entrenamiento del modelo, ya que los ajustes en los pesos de la red son tan pequeños que apenas hay progreso en la optimización, lo que puede llevar a un estancamiento en el rendimiento del modelo.

Por otro lado, también podría ocurrir una explosión del gradiente, en la que los ajustes en los pesos son tan grandes que pueden hacer que el proceso de optimización diverja y que los pesos de la red crezcan exponencialmente, lo que resulta en un rendimiento deficiente del modelo. Estos problemas están relacionados a la rugosidad de la función de pérdida.

Para solucionar este problema, aparecen el tamaño de lote, la tasa de aprendizaje y las funciones de activación. Elegir un tamaño de lote pequeño lleva a más actualizaciones de los pesos, lo que también introduce aleatoriedad al modelo. Por otro lado, lotes más grandes actualizan menos los pesos y pueden ayudar a que el modelo se estabilice más rápido. Respecto al ritmo de aprendizaje, valores menores de esta tasa llevan a una convergencia más rápida, mientras que valores más altos producen más volatilidad. En el caso de las funciones de activación, las *ReLU* y las *Leaky ReLU* son una forma de introducir no linealidad al modelo y de lograr que el gradiente no se desvanezca.

En tercer lugar, el sobreajuste se presenta cuando el modelo logra predecir muy bien los datos del conjunto de entrenamiento pero no se desempeña satisfactoriamente en datos de validación, esto significa que el modelo no sirve para generalizar, es decir, ha aprendido patrones específicos del conjunto de entrenamiento que no se cumplen necesariamente en datos que el modelo no vio.

Para combatir este problema está el *dropout*, que consiste en desactivar una fracción de las conexiones sinápticas durante el entrenamiento. Además, elegir una cantidad de épocas adecuada es importante para prevenir el sobreajuste, ya que un número elevado de épocas en el entrenamiento puede provocar este problema. El *pooling* también está relacionado indirectamente a esta cuestión, en el sentido de que reduce la dimensionalidad de la salida, lo que puede ayudar a simplificar y generalizar el modelo, ya que extrae las características más importantes y descarta la información redundante. Esto contribuye a prevenir el sobreajuste al reducir la complejidad del modelo y mejorar su capacidad para generalizar a datos no vistos.

Por último, hay un aspecto relevante sobre las arquitecturas neuronales: la cantidad de acoplamientos sinápticos. El crecimiento de acoplamientos aumenta exponencialmente la cantidad de mínimos locales y puntos de ensilladura, lo que empeora el desvanecimiento del gradiente. También, cuantos más pesos tenga el modelo, mayor será su capacidad para memorizar el conjunto de entrenamiento, lo que aumenta el riesgo de sobreajuste. Estos aspectos deben considerarse a la hora de elegir el tamaño de la capa oculta. Además, en este trabajo es pertinente tener en cuenta que las redes *feed-forward* suelen tener más acoplamientos que las convolucionales, que además cuentan con el *pooling* como una estatregia para disminuir la cantidad de parámetros.

La siguiente sección presenta los resultados de las implementaciones y detalla las combinaciones de hiperparámetros, esto permitirá comprender el carácter práctico de esta cuestión.

IV. Resultados

Autoencoders

La figura 5 presenta el promedio del *Mean Squared Error* (MSE) por lote para cada época, de los *autoencoders* implementados. En la parte superior, se exponen los errores de entrenamiento y validación para los *autoencoders* convolucional «consigna» y el *autoencoder* lineal. En esta parte del gráfico, las épocas llegan hasta 60. En la parte inferior, se presentan los *autoencoders* convolucional «Raschka» con 64 y 128 neuronas en la capa oculta, que fueron entrenados durante 17 y 14 épocas, respectivamente. La letra n es el tamaño de la capa oculta, es decir, la compresión de los datos que logran los *encoders*.

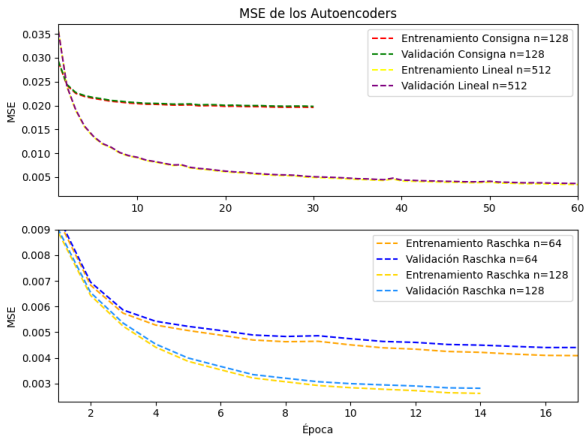

Figura 5. *Mean Squared Error* de los *autoencoders* implementados

El *autoencoder* convolucional «consigna» fue entrenado con un lote de tamaño 32, una tasa de aprendizaje de 0,001 y un *dropout* de 0. El modelo dejó de mejorar significativamente a partir de la época 20. La combinación del tamaño del lote y de la tasa de aprendizaje lograron un entrenamiento estable, no obstante, no se alcanzaron rendimientos satisfactorios. La tasa de *dropout* se bajó a 0 debido a que la brecha entre los errores es casi nula, por lo que no hay sobreajuste. Es pertinente señalar que esta estructura cuenta con funciones de *pooling*.

En el caso del lineal, luego de 60 épocas el modelo continuaba mejorando un poco, pero no significativamente. Este *autoencoder* fue entrenado con un lote de tamaño 1000, una tasa de aprendizaje de 0,001 y un *dropout* de 0,1. El tamaño de la capa oculta fue de 512. La elección de esta combinación de hiperparámetros se hereda del trabajo práctico tres. En la figura 5 se observa que el error desciende de manera continua y estable a lo largo de las épocas.

Respecto a los *autoencoders* convolucionales «Raschka» con 64 y 128 neuronas en la capa oculta, ambos modelos fueron entrenados con un lote de tamaño 32, una tasa de aprendizaje de 0,0005 y un *dropout* de 0. Para determinar el número de épocas se utilizó un método *early stopping*, para prevenir el sobreajuste. Este método consiste en frenar el entrenamiento cuando el error deja de mejorar significativamente o la diferencia entre los errores de validación y entrenameinto comienza a crecer. El modelo con $n = 64$ presenta un rendimiento inferior al modelo de $n = 128$, además el primero se entrenó durante 17 épocas, mientras el segundo necesitó solo 14. En estos *autoencoders*, el pequeño tamaño de lote logró bajar exitosamente el error a niveles muy bajos en pocas épocas, mientras el ritmo de aprendizaje más pequeño alcanzó un descenso del error estable. Se decidió usar una tasa de *dropout* nula, ya que los errores de validación no superan significativamente a los errores de entrenamiento y continúan descendiendo a lo largo de las épocas, mientras que tasas mayores de *dropout* perjudicaron al rendimiento del modelo.

La tabla I resume los distintos modelos y sus correspondientes hiperparámetros.

Tabla I
Los autoencoders y sus hiperparámetros

Modelo	Capa oculta	Épocas	Lote	Tasa de aprendizaje	Dropout
Consigna	128	20	32	0.001	0
Lineal	512	60	1000	0.001	0.1
Raschka (64)	64	17	32	0.0005	0
Raschka (128)	128	14	32	0.0005	0

Los *autoencoders* con mejor desempeño fueron el lineal y el «Raschka» con $n = 128$. El *autoencoder* convolucional logró comprimir las entradas mejor que el lineal, alcanzar errores menores y con un menor costo computacional, ya que lo logró en 14 épocas, en cambio, el *feed-forward* recíen logró un error comparable (ligeramente superior) luego de 60 épocas. También el «Raschka» con $n = 64$ merece una mención honorifíca, en tan solo 17 épocas logró la mayor compresión de los datos y un error muy bajo.

La figura 6 presenta tres imágenes del conjunto de validación y sus correspondientes reconstrucciones por el «Raschka» con $n = 128$ y el lineal con $n = 512$.

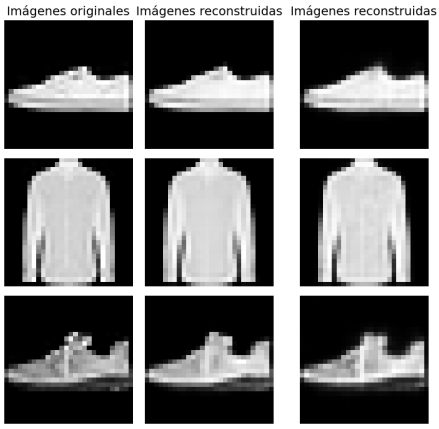

Figura 6. Imágenes reconstruídas por los *autoencoders* ganadores. De izquierda a derecha: imágenes orginales, reconstrucción de convolucional «Raschka» $n = 128$ y reconstrucción de *feed-forward* $n = 512$

Clasificadores

A continuación se muestran los resultados de los clasificadores presentados en la sección anterior. Adicionalmente, se implementó *feature extraction* sobre el clasificador «consigna», ya que se lo inicializó con los pesos del *encoder* entrenado en la sección anterior.

También se intentó dejar fijos los pesos del *encoder* pre-entrenado y optimizar solo la capa clasificadora, con los hiperparámetros ajustados. Sin embargo, esta última aplicación no alcanzó resultados satisfactorios.

Además, se intentó iniciar al clasificador con $n = 10$ con los pesos del *encoder* con $n = 128$, pero la implementación no fue beneficiosa.

La tabla II resume los hiperparámetros de cada modelo.

Tabla II
Los clasificadores y sus hiperparámetros

Modelo	Capa oculta	Épocas	Lote	Tasa de aprendizaje	Dropout
Consigna	10	35	32	0.0005	0.8
Raschka	10	15	40	0.0007	0.8
Consigna FE	128	26	32	0.0005	0.8

En la figura 7 se exponen la *Cross Entropy Loss* y la Precisión de los clasificadores. Para todas las aplicaciones ambas métricas

presentan una mejora continúa y estable a lo largo de las épocas, no obstante, presentan una brecha notable entre los rendimientos en el conjunto de entrenamiento y de validación.

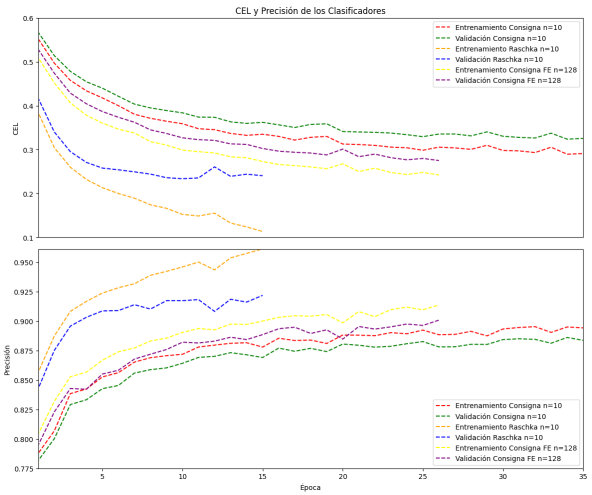

Figura 7. CEL y Precisión de los clasificadores

En el caso de los clasificadores «consigna», la brecha de rendimiento entre los conjuntos no es importante y es constante a lo largo de las épocas. Por otro lado, en el caso del «Raschka» esta brecha crece con las épocas, lo que significa que el modelo presenta señales de sobreajuste, aunque el desempeño en datos no vistos no disminuye significativamente.

Esto último sugiere la incorporación de funciones de pooling en el *encoder* del clasificador «Raschka», para mitigar las señales de sobreajuste, ya que el *dropout* de $0{,}8$ se aplica luego de la función *flatten* y por lo tanto, actúa sobre las capas *fully conected*.

La capa oculta del «consigna» FE (*feature extraction*) es de 128, igual que la del *autoencoder* de la sección anterior. Este modelo presenta mejores resultados que su versión original (con $n = 10$), no obstante, esto podría atribuirse al tamaño de la capa oculta o al *feature extraction*. Para aclarar esta cuestión, una opción sería entrenar al *autoencoder* «consigna» con $n = 10$, realizar esta técnica nuevamente sobre el clasificador «consigna» con $n = 10$ y observar si sus resultados son concluyentes para comprobar si la mejora se debe al tamaño de la capa oculta o al tratamiento de la dependencia de valores iniciales.

Matriz de confusión

La matriz de confusión es una herramienta para evaluar el rendimiento de un modelo de clasificación. Las filas representan a las clases reales y las columnas a las predichas. El clasificador «Raschka» obtuvo la siguiente matriz:

$$\begin{bmatrix} 894 & 2 & 24 & 12 & 3 & 0 & 61 & 0 & 4 & 0 \\ 1 & 986 & 0 & 10 & 0 & 0 & 2 & 0 & 1 & 0 \\ 10 & 1 & 916 & 5 & 36 & 0 & 32 & 0 & 0 & 0 \\ 17 & 2 & 16 & 918 & 14 & 0 & 32 & 0 & 1 & 0 \\ 3 & 0 & 50 & 22 & 891 & 0 & 33 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 983 & 0 & 11 & 1 & 5 \\ 123 & 3 & 71 & 22 & 75 & 0 & 699 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 979 & 0 & 15 \\ 1 & 1 & 4 & 0 & 2 & 1 & 3 & 1 & 987 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 29 & 1 & 964 \end{bmatrix}$$

Dentro de la diagonal principal se observan las instancias correctamente clasificadas, fuera de la diagonal, las incorrectamente clasificadas. Por ejemplo, para la etiqueta 0, el modelo la clasificó 894 veces correctamente, 2 veces como etiqueta 1, 24 veces como etiqueta 2, etc. En la matriz, se observa que el clasificador presenta un gran desempeño para la etiqueta 8, mientras no es tan preciso para la etiqueta 6, que en *Fashion-MNIST* corresponde a la clase *Shirt*.

V. Conclusión

Los resultados de la sección anterior permitieron observar el constraste en las combinaciones óptimas de hiperparámetros de los diferentes modelos.

Sobre la naturaleza del problema, para la reconstrucción de imágenes aumentar el tamaño de la capa oculta logró mejores resultados. En cambio, para la clasificación esta relación no se mantiene, al contrario, los resultados parecen sugerir que un menor tamaño de la capa oculta es más adecuado. Sin embargo, esto no significa que para clasificar imágenes siempre deba usarse un tamaño de capa oculta pequeño, ya que también es importante la capacidad del modelo para capturar la complejidad de los datos. Elegir el tamaño de capa oculta es una cuestión de equilibrio y depende del caso específico que se trate.

Respecto al tipo de neuronas, la red *feed-forward* alcanzó mejores resultados con un tamaño de lote «grande» y una tasa de aprendizaje «alta», mientras que las redes convolucionales aprendieron mejor con lotes «pequeños» y tasas de aprendizaje «bajas». Para combatir el sobreajuste, el *dropout* resultó más efectivo en la red lineal, mientras que el *pooling* fue más adecuado en las convolucionales.

Además, el tipo de neuronas y el tamaño de la capa oculta están relacionados con la cantidad de conexiones sinápticas, que hacen más rugosa la función de pérdida y aumentan el riesgo de sobreajuste. Este detalle explica la diferencia entre los lotes «pequeños» junto con un ritmo de aprendizaje «bajo» de las redes convolucionales, respecto a los lotes «grandes» acompañados de una tasa de aprendizaje «alta» de la red completamente conectada, que tiene más parámetros.

Es pertinente señalar que la influencia de la combinación del tamaño de lote y la tasa de aprendizaje sobre el proceso de entrenamiento puede depender de varios factores, como la complejidad del problema, la arquitectura de la red y la naturaleza de los datos. Por lo tanto, la consideración de la cantidad de acoplamientos es solo una manera de abordar este tema.

En conclusión, elegir los hiperparámetros es una cuestión de carácter teórico-práctico que depende de la naturaleza del problema que se trate y del tipo de redes implementadas. Para este conjunto de datos en particular, las redes convolucionales son superiores a las *feed-forward*, tanto para comprimir y reconstruir imágenes como para clasificarlas. El trabajo de Raschka (2018) [4] es adecuado para estudiar sobre las técnicas de optimización de hiperparámetros.

Referencias

[1] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
[2] Anay Dongre. Overview of Autoencoders. Medium, 2023. Disponible en: https://dongreanay.medium.com/overview-of-autoencoders-52c777418937.
[3] Sebastian Raschka. Convolutional Autoencoder (MNIST). GitHub, 2021. Disponible en: https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/L16/conv-autoencoder_mnist.ipynb.
[4] Raschka, S. (2018, noviembre). Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning. Departamento de Estadística, Universidad de Wisconsin–Madison. Correo electrónico: sraschka@wisc.edu