I302 - Aprendizaje Automático y Aprendizaje Profundo

Trabajo Práctico 3: Redes Neuronales

Agustín Ezequiel Amblard

13 de mayo de 2025

Ingeniería en Inteligencia Artificial

Resumen

En este trabajo se buscó modelar un sistema de clasificación de imágenes manuscritas japonesas utilizando redes neuronales. Para llevarlo a cabo, se implementaron distintos modelos tanto manualmente como con PyTorch, explorando múltiples arquitecturas, técnicas de regularización (como L2 y early stopping) y optimizadores (incluyendo Adam y SGD con mini-batch). Se desarrollaron modelos M0 a M4, donde se aplicaron estrategias de evaluación para comparar el impacto de cada mejora sobre el rendimiento y la capacidad de generalización. El modelo M3, implementado en PyTorch con una arquitectura optimizada de una única capa oculta de 512 neuronas con activación ReLU (`[784, 512, 49]`), resultó ser el más eficaz, alcanzando una precisión del 65.9 % y una pérdida por entropía cruzada de 1.3928 sobre el conjunto de test. Finalmente, utilizando dicho modelo, se generaron las predicciones a posteriori para un conjunto no etiquetado y se almacenaron en un archivo `Amblard_Agustin_predicciones.csv`.

1. Introducción

El reconocimiento automático de caracteres manuscritos es una de las tareas más representativas del aprendizaje supervisado, con aplicaciones en la digitalización de documentos, reconocimiento óptico de caracteres y sistemas de asistencia inteligente. En este trabajo se aborda específicamente la clasificación de caracteres manuscritos japoneses como un problema de clasificación multiclase.

Como punto de partida, se presenta una dataset X_images.npy, el cual partimos en tres sets: Entrenamiento (60 % de los datos), validación (20 % de los datos) y test (20 % de los datos). Esta división del dataset es crucial para la evaluación de cada modelo. La entrada del algoritmo está compuesta por imágenes de caracteres representadas como vectores de 784 características numéricas, correspondientes a una grilla de 28x28 píxeles a escala de grises. Cada imagen pertenece a una de las 49 clases posibles, correspondientes a diferentes símbolos del silabario japonés. En la Figura 1 se pueden apreciar tres imágenes del dataset.

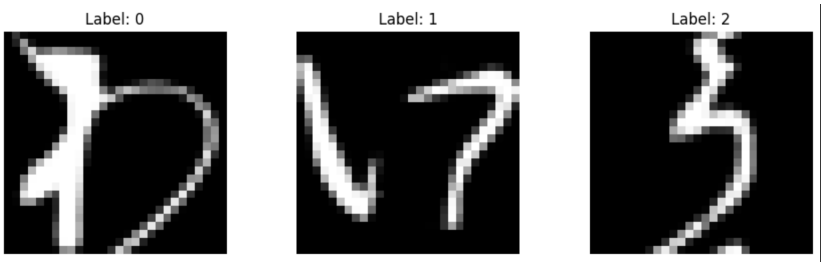

Figura 1: Imagenes pertenecientes al dataset X_images.npy.

La salida esperada del algoritmo es una distribución de probabilidad sobre las 49 clases, indicando la probabilidad a posteriori de que la imagen corresponda a cada una de ellas.

Para resolver el problema, se implementaron distintas redes neuronales profundas (Multilayer Perceptrons) utilizando PyTorch o implementaciones generadas por el alumno, ajustando sus hiperparámetros y arquitectura para mejorar su rendimiento. Se trabajó con técnicas como regularización L2, decaimiento del learning rate, early stopping y optimización mediante el algoritmo ADAM o Stochastic Gradient Descent.

Se implementaron 5 modelos, cada uno con ciertas caracteristicas: El modelo M0 es una red neuronal simple implementada sin librerías de inteligencia artificial, con dos capas ocultas de 100 y 80 nodos respectivamente y usando gradiente descendente estándar sin optimizaciones implementadas. El modelo M1 estaba también implementado sin librerías, pero con la particularidad de que se investigó la combinación de arquitecturas, métodos de optimizacion e hiperparámetros que minimizaran el error sobre el conjunto de validación. Por otro lado, M2 está implementado usando la librería de Pytorch, la cual contiene métodos que estabamos utilizando en los otros modelos. Este modelo es entrenado con la misma arquitectura, hiperparámetros y técnicas de optimización que M1, solo que usando Pytorch. M3 también utiliza Pytorch, y busca la arquitectura de red que menor error tenga. Finalmente, M4 es un modelo overfitteado, el cual tiene un error muy alto a comparación de los otros modelos.

. Por último, se utilizó el modelo con mejor desempeño para generar predicciones a posteriori sobre un conjunto adicional de datos no etiquetados, exportando los resultados en un archivo CSV.

2. Métodos

2.1. Modelo y funcionamiento general

En este trabajo se implementó un clasificador basado en una red neuronal multicapa (*Multilayer Perceptron*) para abordar un problema de clasificación multiclase de 49 clases. La red recibe como entrada un vector de 784 características numéricas, correspondiente a imágenes de 28×28 píxeles a escala de grises, y entrega como salida una distribución de probabilidades sobre las 49 clases posibles.

La arquitectura general consiste en capas densamente conectadas con funciones de activación *ReLU* y una capa de salida con activación *Softmax*. La red fue entrenada mediante backpropagation, utilizando diferentes técnicas de optimización y regularización.

2.2. Funciones de activación y salida

ReLU (Rectified Linear Unit) es utilizada como función de activación no lineal en las capas ocultas:

$$\text{ReLU}(z) = \text{máx}(0, z)$$

Su derivada, utilizada durante backpropagation, es:

$$\frac{d}{dz}\text{ReLU}(z) = \begin{cases} 1 & \text{si } z > 0 \\ 0 & \text{en otro caso} \end{cases}$$

Softmax es empleada en la capa de salida para producir una distribución de probabilidad:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

donde $K = 49$ es la cantidad de clases.

2.3. Función de costo y métricas

Se utilizó la función de pérdida de entropía cruzada (cross-entropy) como medida del error:

$$\mathcal{L}(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log(\hat{y}_k)$$

donde y es el vector codificado one-hot con la clase verdadera, y $\hat{y}$ es la predicción de la red.

Las métricas utilizadas para evaluar el rendimiento fueron:

- **Accuracy**: proporción de predicciones correctas sobre el total.

$$\text{Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(\hat{y}_i = y_i)$$

  donde N es la cantidad total de ejemplos en el conjunto de prueba, y_i es la clase verdadera, $\hat{y}_i$ es la clase predicha por el modelo, y $\mathbb{1}(\cdot)$ es la función indicadora.

- **Matriz de confusión**: Esta herramienta permite visualizar la cantidad de verdaderos positivos, falsos negativos y falsos positivos para cada clase, siendo especialmente útil en problemas con muchas categorías, como este caso con 49 clases.

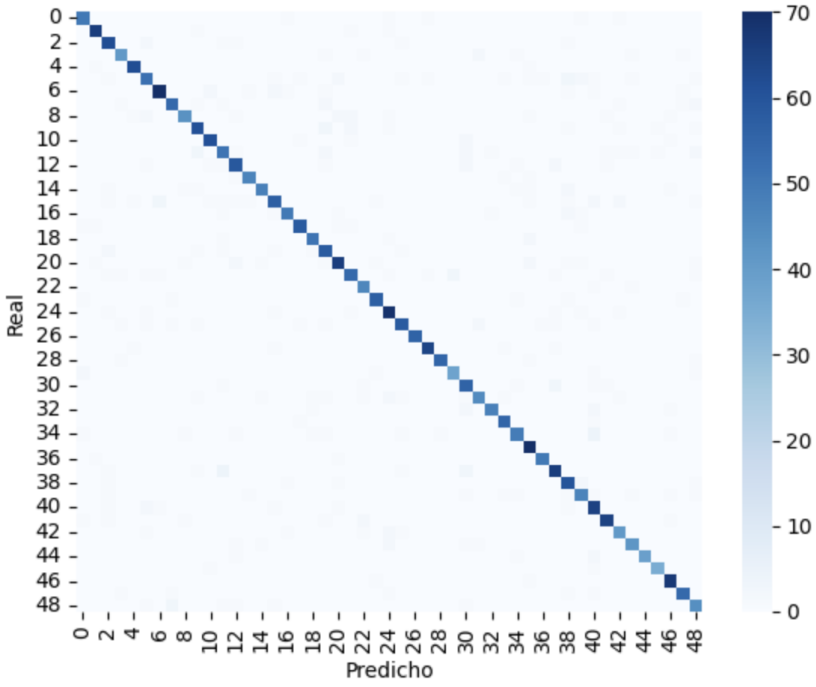

Figura 2: Matriz de confusión.

2.4. Algoritmos de optimización

Se exploraron tres variantes del algoritmo de descenso por gradiente:

- **Gradient Descent (GD)**: El algoritmo de descenso por gradiente busca minimizar una función de pérdida $\mathcal{L}(\theta)$ actualizando los parámetros del modelo θ en la dirección opuesta al gradiente del error con respecto a esos parámetros. La regla de actualización general es:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}(\theta)$$

donde:

- θ representa los pesos y sesgos del modelo,
- η es la tasa de aprendizaje (learning rate),
- $\nabla_\theta \mathcal{L}(\theta)$ es el gradiente de la función de pérdida respecto a los parámetros.

En el caso del descenso por gradiente clásico, este cálculo se realiza sobre todo el conjunto de entrenamiento, lo cual puede ser computacionalmente costoso en grandes datasets.

- **Mini-Batch Stochastic Gradient Descent (Mini-batch SGD)**: SGD introduce una mejora sobre el GD tradicional, actualizando los parámetros utilizando únicamente un subconjunto del conjunto de entrenamiento (mini-batch) en cada paso, reduciendo así el tiempo de cómputo por iteración.

Para un mini-batch $B = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ de tamaño m, la actualización es:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}\left(\theta; x^{(i)}, y^{(i)}\right)$$

Donde $\mathcal{L}\left(\theta; x^{(i)}, y^{(i)}\right)$ es la pérdida para la muestra i del mini-batch. Este enfoque introduce ruido en la estimación del gradiente, lo que puede ayudar a escapar de mínimos locales y explorar mejor la superficie de pérdida.

Además, en cada época se barajan aleatoriamente los datos, lo que mejora la generalización del modelo.

- **Adam**: Adam es un algoritmo de optimización que Utiliza estimaciones adaptativas de primer y segundo momento del gradiente para ajustar el learning rate de cada parámetro.

Sea $g_t = \nabla_\theta \mathcal{L}(\theta_t)$ el gradiente de la función de pérdida respecto a los parámetros θ en el paso t:

1. **Cálculo del primer momento (media exponencial de los gradientes)**:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

2. **Cálculo del segundo momento (media exponencial del cuadrado del gradiente)**:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. **Corrección de sesgo para ambos momentos (bias correction)**:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. **Actualización de parámetros**:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Donde:

- η es el learning rate inicial,
- β_1 y β_2 son coeficientes de decaimiento para los promedios móviles, típicamente $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$,
- ϵ es un pequeño valor constante (por ejemplo, 10^{-8}) para evitar divisiones por cero,
- m_t y v_t son vectores del mismo tamaño que θ, actualizados en cada paso.

Adam es especialmente útil para problemas de alta dimensión y con gradientes escasos, ya que ajusta dinámicamente la magnitud del paso para cada parámetro.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

2.5. Rate scheduling y early stopping

Se implementaron dos tipos de programación del learning rate:

- **Rate scheduling lineal**: el learning rate decae de forma lineal:

$$\eta_t = \text{máx}\left(\eta_0 \left(1 - \frac{t}{T}\right), \eta_{\text{mín}}\right)$$

- **Rate scheduling exponencial**: el learning rate decae exponencialmente:

$$\eta_t = \eta_0 \cdot \gamma^t, \quad \gamma = \left(\frac{\eta_{\text{mín}}}{\eta_0}\right)^{1/T}$$

También se utilizó **early stopping** para evitar el sobreajuste. Early stopping es una técnica de regularización que detiene el entrenamiento del modelo cuando la performance en el conjunto de validación deja de mejorar, lo cual ayuda a prevenir el sobreajuste. En lugar de entrenar durante un número fijo de épocas, el entrenamiento se interrumpe cuando la métrica de validación (por ejemplo, la función de pérdida) no mejora durante p épocas consecutivas, conocido como *paciencia*.

Formalmente, sea $\mathcal{L}_{\text{val}}^{(t)}$ la pérdida de validación en la época t, y t^* la mejor época observada hasta el momento. Se define:

$$t^* = \arg \min_{0 \leq k \leq t} \mathcal{L}_{\text{val}}^{(k)}$$

Entonces, si en las últimas p épocas se cumple que:

$$\mathcal{L}_{\text{val}}^{(t-i)} \geq \mathcal{L}_{\text{val}}^{(t^*)}, \quad \text{para todo } i \in \{0, 1, \ldots, p-1\}$$

se detiene el entrenamiento en la época t y se restauran los pesos del modelo correspondientes a t^*.

Esta estrategia es especialmente útil cuando el modelo comienza a sobreajustarse el conjunto de entrenamiento, y suele contraarrestarse con técnicas como regularización L2 y decaimiento del learning rate.

2.6. Inicialización de pesos

Se empleó **inicialización He**, recomendada para redes con activaciones ReLU:

$$W_l \sim \mathcal{N}\left(0, \frac{2}{n_l}\right)$$

donde n_l es la cantidad de entradas a la capa l.

2.7. Backpropagation

Durante el entrenamiento, se usó el algoritmo de **backpropagation** para computar los gradientes de la función de pérdida respecto a los parámetros. Para cada capa:

1. Se calcula el error en la salida: $\delta_L = \hat{y} - y$
2. Se retropropaga el error: $\delta_l = (W_{l+1}\delta_{l+1}) \odot f'(Z_l)$
3. Se actualizan pesos y sesgos usando los gradientes:

$$\frac{\partial \mathcal{L}}{\partial W_l} = A_{l-1}^T \delta_l \quad \text{y} \quad \frac{\partial \mathcal{L}}{\partial b_l} = \sum_{i=1}^{m} \delta_l^{(i)}$$

2.8. Conjunto de datos y preprocesamiento

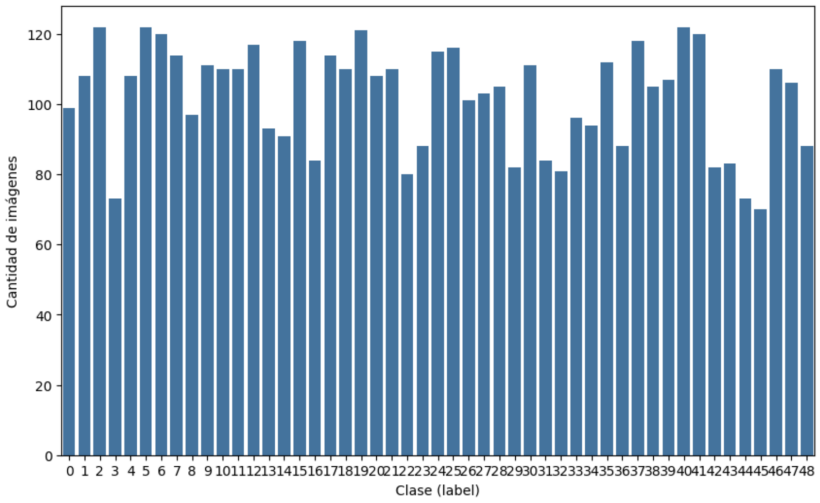

Figura 3: Proporcion de imagenes pertenecientes al dataset.

Se utilizó un conjunto de imágenes vectorizadas de 784 dimensiones, con 49 clases posibles. Primero se estudió la proporción de las clases del dataset. Como podemos observar en la figura 2, se ve que las clases están bastante balanceadas, por lo que se utiliza una división 60 % / 20 % para entrenamiento y validación. El conjunto de test consiste del 20 % restante. Como preprocesamiento, se normalizaron los valores de píxel dividiendo por 255, de modo que los valores estén en el rango $[0, 1]$.

2.9. Selección de hiperparámetros

Para seleccionar los mejores parámetros para cada caso, se realizó un proceso de evaluación por separado de cada optimización de el modelo. Luego de evaluar un modelo por optimización posible, nos quedamos con los valores que mejor minimizaban el error de validación. Los principales hiperparámetros utilizados en los modelos fueron:

- **Learning rate**: se exploraron valores entre 0.0001 y 0.1 según el optimizador.
- **Tamaño de batch**: se utilizaron valores 32 y 64.
- **Arquitectura**: se evaluaron múltiples configuraciones, variando la cantidad de capas ocultas y de nodos en cada una de ellas.
- **Regularización L2**: se ajustó λ en el rango [0.001, 0.1].
- **Paciencia para early stopping**: típicamente 20.
- **Epochs**: 500 para M0, el resto fueron evaluadas en 200 epochs. Esta decisión fue tomada debido a que GD necesita mas iteraciones para converger que los modelos mas optimizados.

La selección se realizó empíricamente evaluando el desempeño en validación.

2.10. Predicciones a posteriori

El archivo `Amblard_Agustin_predicciones.csv` contiene, para cada una de las 1000 muestras del conjunto `X_COMP.npy`, un vector de 49 valores que representan las **probabilidades a posteriori** de pertenecer a cada una de las clases posibles.

Estas probabilidades, denotadas como $P(y = c \mid \mathbf{x})$ para $c \in \{0, 1, \ldots, 48\}$, surgen como salida de la función `softmax` aplicada a la última capa de la red neuronal. Matemáticamente, se definen como:

$$P(y = c \mid \mathbf{x}) = \frac{\exp(z_c)}{\sum_{j=0}^{48} \exp(z_j)}$$

donde z_c es el logit (la salida lineal sin activar) correspondiente a la clase c para la muestra de entrada $\mathbf{x}$. La función softmax transforma estos logits en una distribución de probabilidad sobre las clases, asegurando que:

$$\sum_{c=0}^{48} P(y = c \mid \mathbf{x}) = 1 \quad \text{y} \quad 0 \leq P(y = c \mid \mathbf{x}) \leq 1$$

En este contexto, el modelo estima para cada muestra cuál es la probabilidad de pertenecer a cada clase, dado lo aprendido durante el entrenamiento. Esta salida probabilística es útil tanto para clasificaciones determinísticas (eligiendo la clase de mayor probabilidad) como para análisis más finos, como cálculo de entropía, incertidumbre o calibración.

Por ejemplo, si una fila del archivo tiene un valor de 0,85 en la columna `Clase_7`, esto significa que el modelo asigna una probabilidad del 85 % a que esa muestra pertenezca a la clase 7.

3. Resultados

En esta sección se presentan los resultados obtenidos tras entrenar múltiples modelos con distintas configuraciones de arquitectura y optimización. Se evaluaron cinco modelos principales (denominados M0 a M4), con variaciones en la cantidad de capas, regularización, técnica de optimización y programación del learning rate.

3.1. Comparación de desempeño

A continuación, se resume el rendimiento de los modelos evaluados sobre el conjunto de test:

Modelo	Accuracy	Loss	Técnicas	Observaciones
M0	0.5780	1.7355	GD	lr = 0.1
M1	0.6530	1.7047	Adam(batch_size=32), L2.	lr = 0.001, lambda_l2 =0.001
M2	0.6590	1.6415	Adam(batch_size=32), L2	lr = 0.001, lambda_l2 =0.001
M3	**0.6590**	**1.3928**	Adam(batch_size=32), L2	lr = 0.001, lambda_l2 = 0.001
M4	0.5930	2.8300	Adam sin regularización	lr = 0.001 y arquitectura muy grande

Cuadro 1: Comparación de rendimiento en test

M0: Utilizó gradiente descendente estándar (GD) con learning rate fijo de 0,1, sin técnicas de regularización ni adaptación del learning rate. Si bien logró una precisión de entrenamiento relativamente alta (88.7 %), su desempeño en validación cayó significativamente (58.4 % de accuracy, con pérdida de 1.6954). Esto evidencia un **leve overfitting** y una limitada capacidad de generalización. Además, el optimizador GD no es tan eficiente como otros métodos más modernos como Adam, especialmente en problemas con muchas clases como este.

M1: Este modelo usó Adam con batch size 32 y regularización L2 ($\lambda = 0{,}001$). Obtuvo un val_loss mínimo de 1.5045 y una val_acc máxima de 68.2 %. Comparado con M0, representa una mejora clara tanto en pérdida como en precisión de validación. Esto demuestra que el uso de un optimizador más sofisticado y regularización ayudaron a mejorar la generalización del modelo.

M2: También empleó Adam con L2, pero con la implementación con Pytorch. Alcanzó una accuracy de validación de 65.1 % y una pérdida de 1.6102. Aunque su desempeño es inferior al de M1 en este conjunto, sigue siendo superior al de M0. Esto sugiere que la estructura usada

en M2 no fue tan eficiente para capturar las regularidades del problema como en M1, aunque las técnicas de optimización fueron adecuadas.

M3: Fue el modelo con mejor desempeño en términos de pérdida de validación: 1.3077. También mostró un accuracy de validación elevado (67.3 %) y una pérdida de entrenamiento baja (0.0423), con accuracy perfecta (1.000) en entrenamiento. Esto indica un posible principio de **overfitting leve**, pero el gap entre entrenamiento y validación no es tan drástico. Se trata del mejor modelo en términos de compromiso entre poder predictivo y generalización.

En la Figura 4 se presentan las curvas de *loss* y *accuracy* para el modelo M3 durante las 200 épocas de entrenamiento.

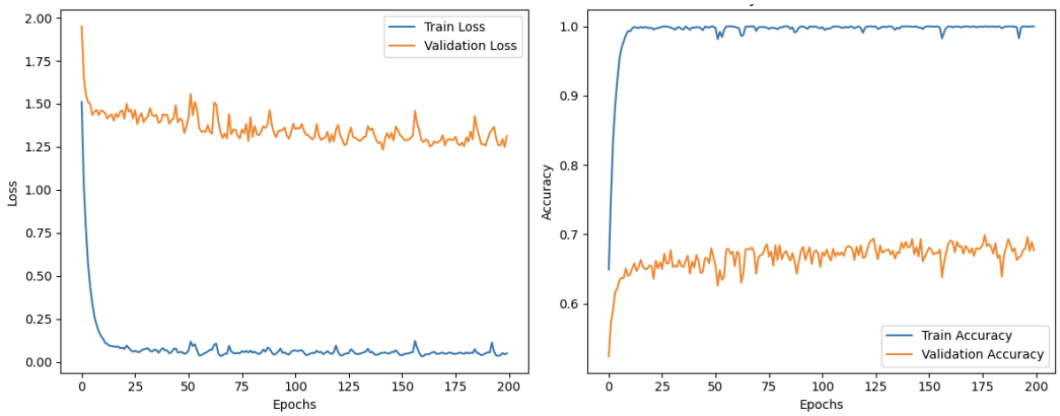

Figura 4: Curvas de entrenamiento y validación para el modelo M3

En el gráfico de la izquierda se observa que la **función de pérdida en entrenamiento** (*train loss*) desciende rápidamente durante las primeras épocas y se estabiliza en valores cercanos a cero, lo que indica que el modelo logra un ajuste casi perfecto sobre el conjunto de entrenamiento. Sin embargo, la **pérdida en validación** (*validation loss*) disminuye inicialmente pero luego se mantiene relativamente constante con oscilaciones, lo cual refleja una brecha entre entrenamiento y validación característica del **sobreajuste**.

El gráfico de la derecha muestra la **precisión en entrenamiento** que alcanza rápidamente un valor cercano a 1, mientras que la **precisión en validación** se estabiliza alrededor de 0.67. Esta diferencia también evidencia que el modelo presenta un leve a moderado *overfitting*: logra memorizar perfectamente los ejemplos de entrenamiento, pero su capacidad de generalización es limitada.

Este comportamiento puede atribuirse a una arquitectura con alta capacidad (512 neuronas ocultas), combinada con un conjunto de datos de tamaño fijo sin técnicas adicionales de regularización más allá de L2. No obstante, el rendimiento sobre el conjunto de validación se mantiene estable, lo que sugiere que el modelo aún logra una generalización razonable.

M4: Utilizó una arquitectura muy grande, sin regularización. Alcanzó loss de entrenamiento nulo y accuracy de 1.0000, pero su pérdida en validación se disparó hasta 7.3719. Esto representa un **overfitting extremo**: el modelo memorizó el set de entrenamiento y perdió toda capacidad de generalización. Esto justifica la necesidad de técnicas como L2 o early stopping al trabajar con redes profundas.

3.2. Conclusión

A lo largo de este trabajo se exploraron múltiples arquitecturas y configuraciones de entrenamiento para resolver un problema de clasificación multiclase con 49 categorías. Se compararon diferentes técnicas de optimización, regularización y estructuras de red.

Los resultados obtenidos evidencian que el uso de optimizadores avanzados como **Adam**, combinado con **regularización L2**, permite alcanzar un buen equilibrio entre sesgo y varianza, favoreciendo la capacidad de generalización. En particular, el modelo **M3**, con arquitectura [784, 512, 49], logró el mejor desempeño, destacándose por su baja pérdida en validación y alta precisión. Este resultado sugiere que una red moderadamente profunda, con técnicas adecuadas de regularización y optimización, es capaz de resolver con eficacia tareas como esta.

No obstante, se observó que **todos los modelos presentaron cierto grado de sobreajuste**. Este fenómeno se manifestó en una precisión muy alta en entrenamiento (incluso perfecta en algunos casos), pero con una caída notable en validación. Esto puede explicarse por varios factores: la capacidad expresiva de las redes empleadas (con muchas neuronas), la ausencia de técnicas como *data augmentation*, y el hecho de trabajar con un número limitado de muestras frente a una gran cantidad de clases.

A pesar del sobreajuste, el rendimiento en validación y test fue razonablemente alto en varios modelos. Esto indica que el **overfitting estuvo contenido**, en parte gracias al uso de regularización L2, al empleo de mini-batch training y al optimizador Adam, que favorece una convergencia más robusta. En resumen, los mejores resultados se obtuvieron al combinar redes suficientemente expresivas con regularización, sin recurrir a arquitecturas excesivamente profundas.

Se concluye que, en este tipo de tareas, una adecuada elección de hiperparámetros y técnicas de control de complejidad son fundamentales para lograr modelos que no solo aprendan bien, sino que también generalicen con éxito.