

---

# TRABAJO PRÁCTICO: REDES NEURONALES Y APRENDIZAJE PROFUNDO

---

**Tobio Santiago**

Departamento de ingeniería en Inteligencia Artificial  
Universidad de San Andres  
Buenos Aires, Argentina  
stobio@udesa.edu.ar

October 30, 2025

## ABSTRACT

Este trabajo presenta la implementación desde cero de un sistema completo de entrenamiento de redes neuronales multicapa para clasificación de caracteres manuscritos en el dataset EMNIST Bymerge (809,555 imágenes, 47 clases con desbalance 15:1). Se implementaron manualmente todos los componentes fundamentales (forward pass, backpropagation, optimizadores SGD y Adam, early stopping) utilizando CuPy para aceleración GPU. Se realizó una exploración sistemática de 12 configuraciones evaluando capacidad del modelo (112k a 445k parámetros), optimizadores (SGD vs Adam), regularización (L2, early stopping), learning rate scheduling, y arquitecturas (profundas vs anchas). La validación con PyTorch confirmó la correctitud de la implementación custom (diferencia  $<0.3\%$  en test accuracy). El mejor modelo (M3b) incorpora batch normalization, dropout 0.3 y activación SiLU, alcanzando 88.83% accuracy en test (+1.38% sobre baseline) y demostrando robustez excepcional ante ruido gaussiano (87.60% accuracy con  $\sigma = 0.1$  vs 80.04% del baseline). Los hallazgos principales revelan que (i) la capacidad del modelo es el factor más determinante, (ii) Adam supera a SGD por 6.11 puntos porcentuales, (iii) arquitecturas profundas superan ligeramente a anchas con capacidad equivalente, y (iv) batch normalization es crucial tanto para accuracy como para robustez ante perturbaciones.

## 1 Introducción

En este Trabajo Práctico se aborda el desarrollo e implementación de modelos de redes neuronales profundas (MLP's) desde cero en Python. El objetivo principal es comprender los fundamentos teóricos y prácticos detrás de las redes neuronales y evaluar el rendimiento de diferentes arquitecturas y técnicas de optimización sobre esta familia de modelos.

Evaluamos el desempeño de los modelos sobre el dataset de EMNIST Bymerge. Este conjunto de datos es una extensión del clásico MNIST, que incluye imágenes de dígitos manuscritos y letras mayúsculas y minúsculas, proporcionando un desafío adicional para la clasificación de imágenes.

Los modelos reciben como input un vector de 784 dimensiones (28x28 píxeles aplanados) y tienen como output una probabilidad para cada una de las 47 clases posibles (dígitos del 0 al 9, letras mayúsculas A-Z y letras minúsculas a-z, excluyendo algunas letras para evitar confusiones). La clasificación se realiza determinando la clase a la que le corresponde la mayor probabilidad en el output del modelo.

Para acelerar el proceso de entrenamiento y aprovechar el compute de operaciones matrices a través de la GPU, se utilizó la biblioteca CuPy, que ofrece una interfaz similar a NumPy pero con soporte para cálculos en GPU.

Se definieron 5 configuraciones experimentales variando arquitectura, optimizador y técnicas de regularización. El modelo **M0** establece la línea base con 2 capas ocultas [128, 64], activación ReLU, softmax en la salida, pérdida de entropía cruzada, y optimización mediante SGD con batch size 512, learning rate 0.001 y sin momentum. Los modelos **M1** exploran sistemáticamente el espacio de hiperparámetros mediante 8 variantes (M1a-M1h) que evalúan diferentes

optimizadores (SGD con momentum, Adam), técnicas de regularización (L2, early stopping), learning rate scheduling (lineal y exponencial), y arquitecturas alternativas (más profundas y más anchas). El modelo **M2** reimplementa en PyTorch la configuración óptima encontrada en M1, permitiendo validar la implementación custom. Finalmente, los modelos **M3** exploran arquitecturas modernas con PyTorch, incorporando funciones de activación alternativas (GELU, SiLU), batch normalization y dropout.

Posteriormente se presentan los resultados en conjuntos de validación y test, y se evalúa la robustez de los modelos frente a perturbaciones de ruido gaussiano en las imágenes de test.

**Nota:** Se recomienda consultar el Jupyter Notebook `Tobio_Santiago_Notebook_TP3.ipynb` como respaldo técnico de este informe, donde se incluyen las implementaciones completas, visualizaciones interactivas, y resultados detallados de todos los experimentos.

## 2 Exploración y preprocesamiento de datos

El conjunto de datos EMNIST Bymerge contiene 809,555 imágenes de  $28 \times 28$  píxeles distribuidas en 47 clases. La Figura 1 muestra un ejemplo representativo de cada clase, evidenciando la complejidad del problema de clasificación debido a la similitud visual entre ciertos caracteres (por ejemplo, 'O' y '0', 'l' y '1').

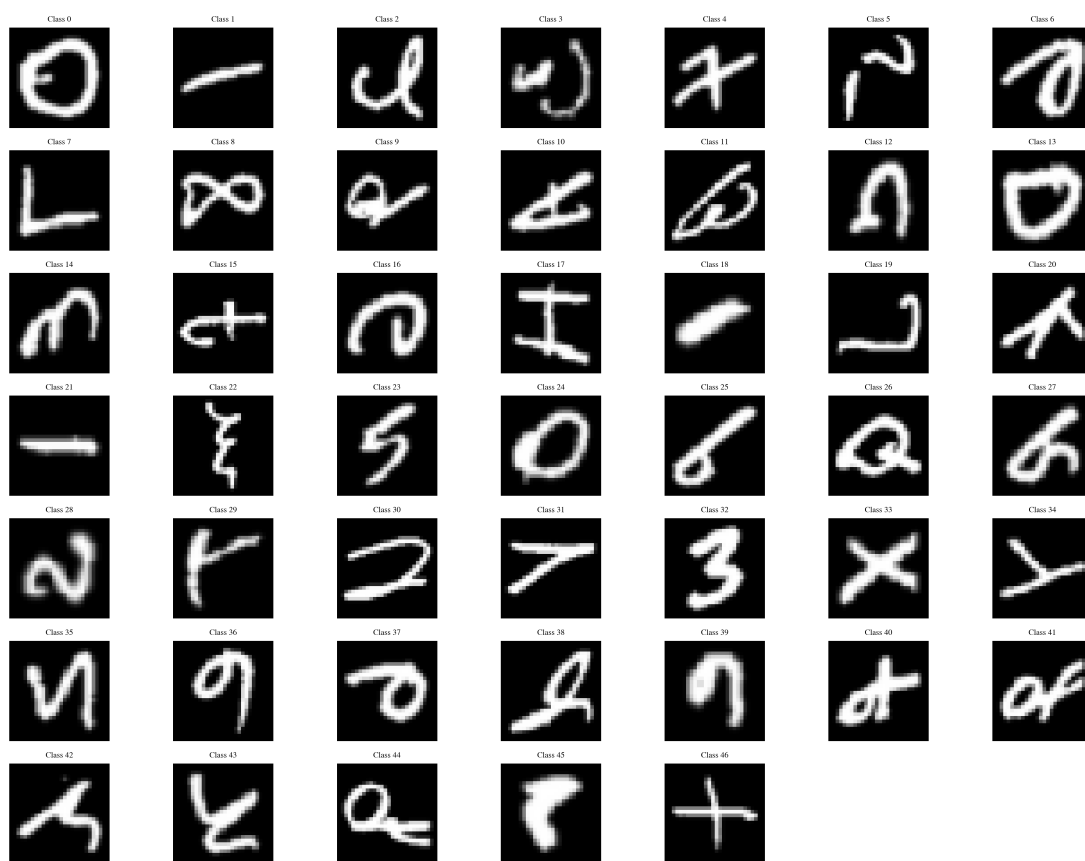


Figure 1: Muestra de una imagen por cada una de las 47 clases del dataset EMNIST Bymerge.

### 2.1 Distribución de clases

El dataset presenta un desbalance significativo entre clases, con una media de 17,225 muestras por clase y una desviación estándar de 13,696. La clase más representada (clase 1) contiene 44,604 muestras, mientras que la menos representada (clase 40) contiene apenas 2,861 muestras, resultando en un ratio de 15.59:1. La Figura 2 ilustra esta distribución heterogénea.

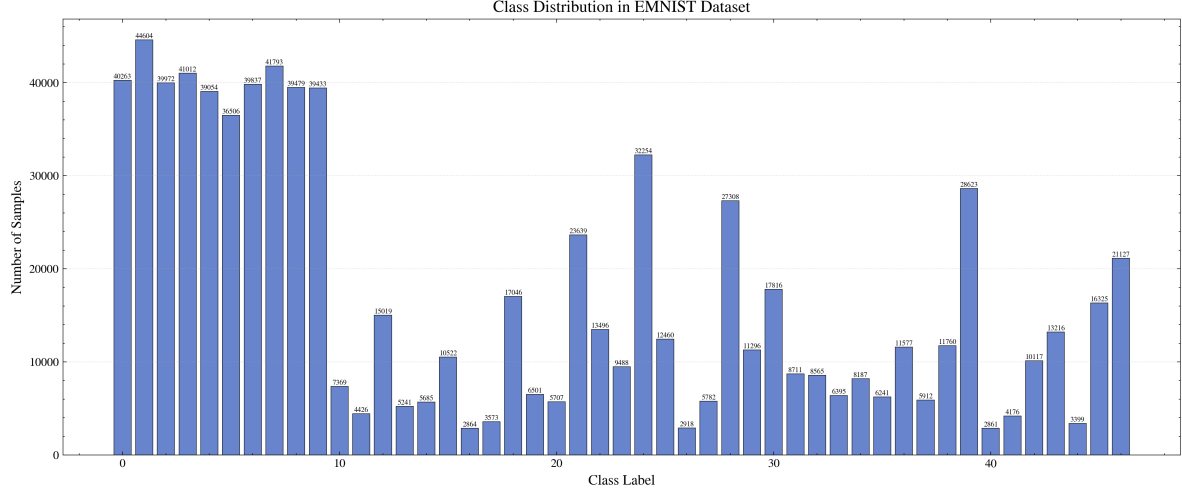


Figure 2: Distribución de muestras por clase en el dataset EMNIST Bymerge, mostrando el desbalance presente en los datos.

## 2.2 Partición estratificada

Dado el desbalance observado, se implementó una partición estratificada que preserva la proporción de clases en cada subconjunto. La división resultante se presenta en la Tabla 1.

Conjunto	Muestras	Proporción	Min/clase	Max/clase
Entrenamiento	566,667	70%	2,002	31,222
Validación	80,933	10%	286	4,460
Test	161,955	20%	573	8,922

Table 1: Partición estratificada del dataset en conjuntos de entrenamiento, validación y test.

La estratificación garantiza que los ratios entre conjuntos se mantienen constantes: Train/Val = 7.00:1, Train/Test = 3.50:1, y Val/Test = 0.50:1. Más importante aún, las proporciones de cada clase se preservan idénticamente en los tres conjuntos, como se verificó comparando la distribución porcentual de las primeras 5 clases (todas con diferencias menores a 0.01% entre conjuntos).

## 2.3 Preprocesamiento

Las imágenes originales (uint8 en rango [0, 255]) se normalizaron dividiéndolas por 255 para obtener valores float32 en el rango [0, 1]. Cada imagen 28×28 se aplanó en un vector de 784 dimensiones, y las etiquetas se codificaron en formato one-hot (vectores de 47 dimensiones). Este preprocesamiento se aplicó de forma consistente en todos los experimentos para garantizar comparabilidad.

## 3 Arquitectura y fundamentos teóricos

Un perceptrón multicapa (MLP) es una composición de funciones vectoriales donde cada capa aplica una transformación afín seguida de una activación no lineal. Para la capa  $\ell$ , la salida se calcula como  $z^{(\ell)} = W^{(\ell)}a^{(\ell-1)} + b^{(\ell)}$  y  $a^{(\ell)} = \sigma(z^{(\ell)})$ , donde  $W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$  es la matriz de pesos,  $b^{(\ell)} \in \mathbb{R}^{n_\ell}$  el vector de sesgos, y  $\sigma$  la función de activación.

Las capas ocultas emplean la función ReLU,  $\text{ReLU}(x) = \max(0, x)$ , que introduce no linealidad permitiendo aproximar funciones complejas mientras su gradiente simple facilita el entrenamiento y mitiga gradientes desvanecientes. La capa de salida utiliza softmax,  $\text{softmax}(z_i) = e^{z_i} / \sum_{j=1}^{47} e^{z_j}$ , que transforma los logits en una distribución de probabilidad sobre las 47 clases.

Para el problema de clasificación multiclase se utiliza la función de pérdida de entropía cruzada categórica,  $\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^{47} y_{ic} \log(\hat{y}_{ic})$ , que mide la discrepancia entre la distribución predicha  $\hat{y}$  y la verdadera  $y$  en formato one-hot. Dado que  $y_i$  es one-hot, esta se simplifica a  $\mathcal{L} = -\frac{1}{N} \sum_{i=1}^N \log(\hat{y}_{i,c_i})$  donde  $c_i$  es la clase verdadera, penalizando fuertemente predicciones con baja probabilidad en la clase correcta.

### 3.1 Algoritmos de optimización

El entrenamiento se realiza mediante descenso de gradiente estocástico (SGD) con mini-batches, donde el gradiente se computa mediante backpropagation. La actualización básica es  $\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \mathcal{L}(\theta_t)$ , donde  $\theta$  representa todos los parámetros del modelo,  $\eta$  es la tasa de aprendizaje, y  $\nabla_{\theta} \mathcal{L}$  el gradiente de la pérdida.

Para acelerar la convergencia, algunos experimentos incorporan **momentum**, que mantiene una velocidad acumulada combinando gradiente actual con previos:  $v_{t+1} = \beta v_t + \nabla_{\theta} \mathcal{L}(\theta_t)$  y  $\theta_{t+1} = \theta_t - \eta v_{t+1}$ , con  $\beta = 0.9$  típicamente. Esto permite atravesar mínimos locales superficiales y converger más rápidamente.

El optimizador **Adam** adapta la tasa de aprendizaje para cada parámetro individualmente, combinando momentum con normalización por magnitud de gradientes. Mantiene estimaciones del primer momento  $m_t$  y segundo momento  $v_t$  del gradiente, corregidas por bias, para actualizar parámetros como  $\theta_{t+1} = \theta_t - \eta \hat{m}_{t+1} / (\sqrt{\hat{v}_{t+1}} + \epsilon)$  con  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  (ver Apéndice B para ecuaciones completas).

En algunos experimentos se aplica **learning rate scheduling** para que el modelo explore ampliamente en épocas tempranas y refine posteriormente. Se implementaron decaimiento lineal ( $\eta_t = \max(\eta_{\min}, \eta_0 - \alpha \cdot t)$ ) y exponencial ( $\eta_t = \eta_0 \cdot \gamma^t$  con  $\gamma \approx 0.95$ ).

### 3.2 Técnicas de regularización

Para prevenir overfitting se emplearon diversas técnicas. La **regularización L2** agrega un término de penalización a la pérdida,  $\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2} \sum_{\ell=1}^L \|W^{(\ell)}\|_F^2$ , que desalienta pesos de magnitud elevada forzando al modelo a distribuir la información entre múltiples pesos. El **early stopping** monitorea la pérdida de validación y detiene el entrenamiento si no mejora durante  $p$  épocas consecutivas, previniendo especialización excesiva en entrenamiento. Los modelos PyTorch adicionalmente exploraron **dropout** (desactivación aleatoria de neuronas durante entrenamiento) y **batch normalization** (normalización de activaciones por mini-batch).

## 4 Metodología de entrenamiento y evaluación

Debido a limitaciones de memoria GPU (NVIDIA RTX 3050 Ti con 4 GB VRAM), se implementó una estrategia híbrida donde el dataset permanece en RAM del sistema y únicamente los mini-batches de 512 muestras se transfieren bajo demanda a GPU para ejecutar forward y backward pass. Esto permite aprovechar la aceleración GPU sin errores de memoria agotada (detalles en Apéndice A). El batch size de 512 se determinó empíricamente como punto óptimo balanceando velocidad de entrenamiento, estabilidad de convergencia y uso de memoria.

### 4.1 Marco experimental unificado

Para garantizar comparaciones justas y reproducibles, todos los experimentos se ejecutan bajo un marco unificado implementado en el módulo `experiments/`. La clase `Dataset` realiza el split estratificado una única vez al inicio, y todos los experimentos subsiguientes reciben referencias a los mismos arrays de train/val/test, eliminando variaciones aleatorias en la partición. Cada experimento se define mediante un objeto `ExperimentConfig` que especifica completamente la arquitectura, hiperparámetros y técnicas de regularización, permitiendo reproducibilidad sin alterar el código. El sistema construye automáticamente todos los componentes (modelo, optimizador, trainer, evaluador) y sigue un flujo de entrenamiento estandarizado implementado en `Trainer`: iteración sobre mini-batches, forward pass, cómputo de pérdida, backpropagation, actualización de parámetros, y evaluación en validación al final de cada época.

### 4.2 Métricas de evaluación

La evaluación de modelos se realiza mediante `evaluate_model` que computa un conjunto fijo de métricas de forma consistente. El **accuracy** mide la proporción de predicciones correctas, aunque puede ser engañoso con clases desbalanceadas. El **F1-Score Macro** promedia el F1 de cada clase sin ponderar por frecuencia, tratando equitativamente la clase con 2,861 muestras y la de 44,604, lo cual es particularmente relevante dado el desbalance del dataset. La **cross-entropy** evalúa no solo corrección sino también confianza en las predicciones, con valores bajos indicando

alta probabilidad en la clase correcta. La **matriz de confusión** de  $47 \times 47$  registra predicciones por clase permitiendo identificar confusiones sistemáticas entre caracteres visualmente similares. Todas las métricas se computan mediante implementación centralizada en `metrics.py`, garantizando que diferencias reflejen genuinamente el rendimiento y no variaciones en el cálculo.

## 5 Resultados experimentales

Esta sección presenta los resultados obtenidos en cada configuración experimental, comenzando por el modelo baseline M0 y posteriormente comparando las variantes M1, M2 y M3.

### 5.1 Modelo M0: Baseline

El modelo M0 establece la línea base de rendimiento contra la cual se comparan las configuraciones posteriores. Su arquitectura consiste en dos capas ocultas de 128 y 64 neuronas respectivamente, con activación ReLU. El entrenamiento se realizó mediante SGD sin momentum ( $\beta = 0$ ) con tasa de aprendizaje constante  $\eta = 0.001$ , batch size de 512, y sin aplicar regularización L2 ni early stopping.

#### 5.1.1 Curvas de entrenamiento

La Figura 3 muestra la evolución de la pérdida y accuracy durante las 50 épocas de entrenamiento. Se observa que la pérdida de entrenamiento decrece consistentemente durante todo el entrenamiento, mientras que la pérdida de validación comienza a incrementarse aproximadamente después de la época 30, señal característica de overfitting. El accuracy de entrenamiento continúa mejorando hasta alcanzar valores cercanos a 90%, mientras que el accuracy de validación se estabiliza alrededor de 87% y muestra ligeras oscilaciones en las épocas finales.

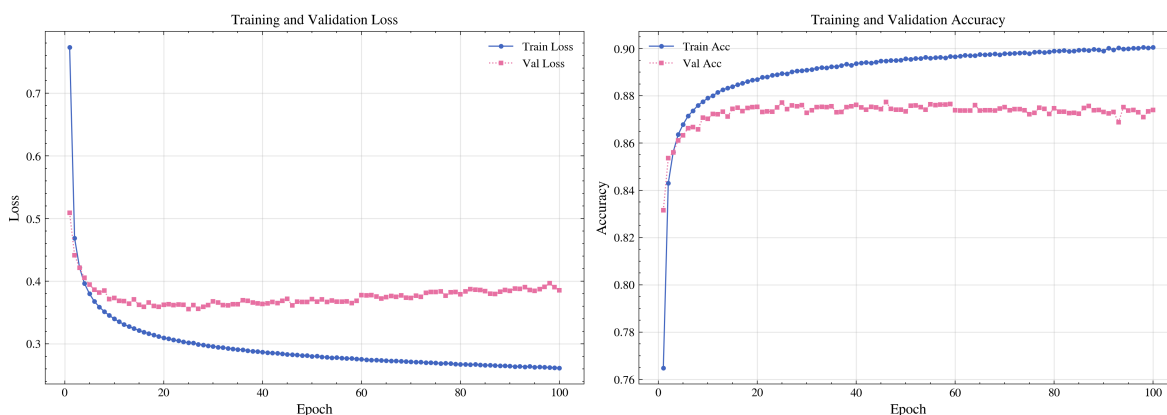


Figure 3: Curvas de entrenamiento del modelo M0. Izquierda: evolución de la pérdida en train y validación. Derecha: evolución del accuracy. Se observa overfitting a partir de la época 30, evidenciado por el incremento de la pérdida de validación mientras la de entrenamiento continúa decreciendo.

La divergencia entre las curvas de pérdida después de la época 30 evidencia overfitting claro: la pérdida de entrenamiento continúa decreciendo mientras que la de validación incrementa, indicando que el modelo comienza a memorizar el conjunto de entrenamiento a costa de generalización. El gap final entre train accuracy (90%) y validation accuracy (87%) de aproximadamente 3 puntos porcentuales confirma este comportamiento. Este overfitting es esperado dada la ausencia de técnicas de regularización (L2, dropout, early stopping) en M0. El modelo habría beneficiado de detener el entrenamiento alrededor de la época 30-35, cuando la pérdida de validación alcanza su mínimo.

#### 5.1.2 Matriz de confusión

La Figura 4 presenta la matriz de confusión del modelo M0 sobre el conjunto de test. La diagonal principal concentra la mayoría de las predicciones, indicando que el modelo generaliza razonablemente bien. Sin embargo, se observan bandas de confusión sistemática en ciertas regiones de la matriz, correspondientes a caracteres visualmente similares.

Algunos patrones de error notables incluyen confusiones entre dígitos y letras que comparten forma visual (por ejemplo, '0' vs 'O', '1' vs 'l' o 'I'), así como entre letras mayúsculas y minúsculas de forma similar. Estos errores son inherentes a la naturaleza del problema y reflejan ambigüedades genuinas en los datos.

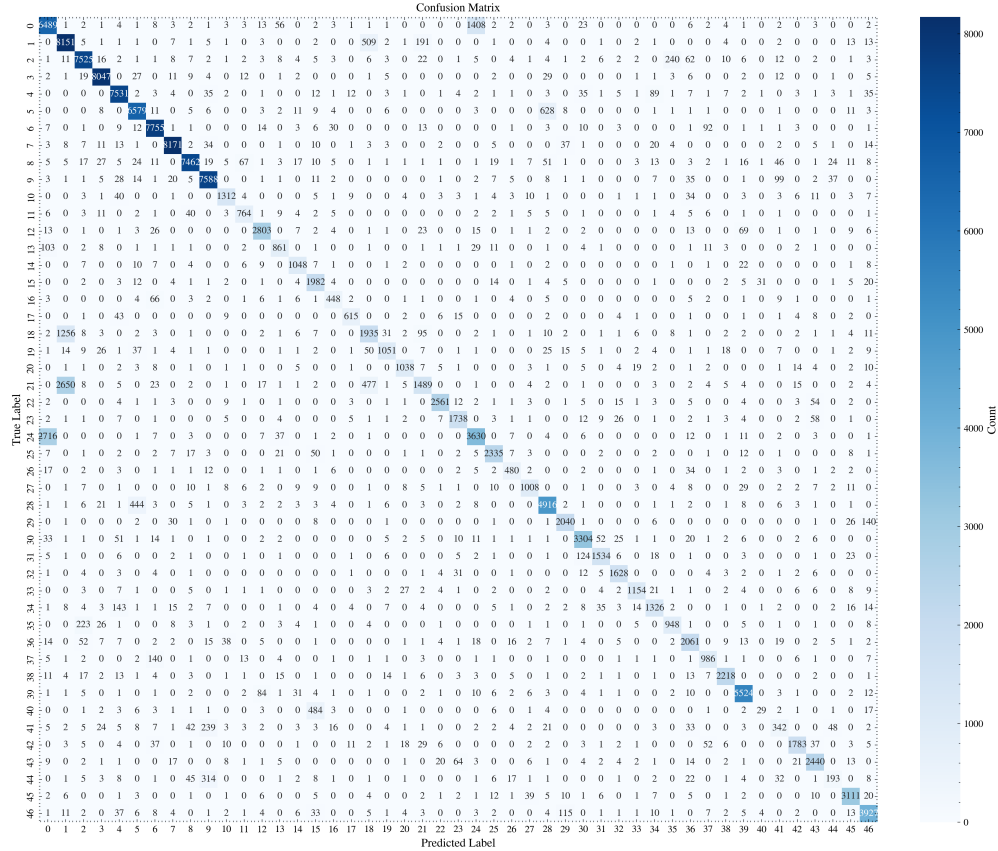


Figure 4: Matriz de confusión del modelo M0 sobre el conjunto de test. Las regiones fuera de la diagonal revelan confusiones sistemáticas entre caracteres similares.

### 5.1.3 Análisis

El modelo M0 establece un baseline sólido, demostrando que una arquitectura simple de MLP con dos capas ocultas es capaz de aprender representaciones efectivas del dataset EMNIST Bymerge, validando la viabilidad de la implementación y la estrategia de gestión de memoria GPU.

No obstante, el overfitting observado a partir de la época 30 y los patrones de confusión sistemática entre caracteres visualmente similares sugieren claras oportunidades de mejora. Los experimentos M1 exploran técnicas de regularización (L2, early stopping), optimizadores avanzados (Adam), learning rate scheduling, y arquitecturas con mayor capacidad para abordar estas limitaciones. Los resultados cuantitativos en el conjunto de test se reportan en la comparación final (Sección 4.5).

## 5.2 Modelo M1: Búsqueda de hiperparámetros

Para mejorar el rendimiento del modelo baseline M0, se realizó una exploración sistemática de hiperparámetros organizando 8 variantes (M1a-M1h) en 4 grupos experimentales. Cada grupo evalúa una dimensión específica del espacio de hiperparámetros manteniendo los demás factores constantes, permitiendo aislar el efecto de cada técnica.

### 5.2.1 Diseño experimental

Los grupos experimentales se definieron como sigue:

**Grupo 1: Optimizadores.** Se compara SGD con momentum ( $\beta = 0.9$ ) contra Adam, ambos con early stopping (patience=5). Este grupo evalúa si optimizadores adaptativos ofrecen ventajas sobre métodos de momentum tradicionales en este problema.

**Grupo 2: Regularización.** Se evalúa el efecto de regularización L2 ( $\lambda = 0.01$ ) combinada con diferentes optimizadores. M1c utiliza Adam + L2, mientras M1d combina SGD con momentum + L2 + early stopping. Este grupo determina si la regularización explícita es necesaria dado el uso de early stopping.

**Grupo 3: Learning Rate Scheduling.** Se exploran dos estrategias de decaimiento de tasa de aprendizaje: lineal (M1e) y exponencial (M1f), ambas con Adam como optimizador base. Este grupo evalúa si el ajuste dinámico del learning rate mejora la convergencia más allá de la adaptación automática de Adam.

**Grupo 4: Arquitectura.** Se contrastan dos aproximaciones para incrementar la capacidad del modelo: una red más profunda con 3 capas [400, 240, 120] totalizando 444,847 parámetros (M1g), versus una red más ancha con 2 capas [370, 370] totalizando 445,157 parámetros (M1h). Ambas arquitecturas tienen aproximadamente la misma cantidad de parámetros (445k), permitiendo comparar profundidad vs anchura con capacidad equivalente.

## 5.2.2 Resultados

La Tabla 2 presenta los resultados completos de las 8 variantes M1 evaluadas sobre los conjuntos de entrenamiento y validación. El mejor modelo según accuracy de validación es M1g (88.51%), seguido por M1h (88.23%). Notablemente, M1g también alcanza la menor pérdida de validación (0.3322), indicando predicciones mejor calibradas.

Modelo	Grupo	Arquitectura	Train Acc	Val Acc	Val Loss	Parámetros
M1a	Opt.	[128, 64]	83.15	81.41	0.5815	111,791
M1b	Opt.	[128, 64]	89.29	87.52	0.3640	111,791
M1c	Reg.	[128, 64]	89.96	87.89	0.3555	111,791
M1d	Reg.	[128, 64]	84.49	84.30	0.4933	111,791
M1e	Sched.	[128, 64]	89.53	87.64	0.3630	111,791
M1f	Sched.	[128, 64]	89.35	87.67	0.3583	111,791
M1g	Arch.	[400, 240, 120]	<b>90.66</b>	<b>88.51</b>	<b>0.3322</b>	444,847
M1h	Arch.	[370, 370]	90.68	88.23	0.3374	445,157

Table 2: Resultados de las variantes M1 en conjuntos de entrenamiento y validación. El mejor modelo por validación es M1g con 88.51% accuracy y pérdida de 0.3322.

## 5.2.3 Análisis por grupo experimental

**Optimizadores (M1a vs M1b):** Adam (M1b: 87.52% val acc) superó significativamente a SGD con momentum (M1a: 81.41% val acc), demostrando la ventaja de tasas de aprendizaje adaptativas por parámetro. La diferencia de 6.11 puntos porcentuales valida que Adam es esencial para convergencia eficiente en este problema. Adicionalmente, Adam alcanza accuracy de entrenamiento considerablemente mayor (89.29% vs 83.15%), indicando que SGD con momentum tiene dificultades para optimizar incluso el conjunto de entrenamiento.

**Regularización (M1c vs M1d):** La regularización L2 combinada con Adam (M1c) mejoró marginalmente el rendimiento (87.89% val acc vs 87.52% de M1b sin L2), con una pérdida de validación ligeramente menor (0.3555 vs 0.3640), sugiriendo predicciones mejor calibradas. Por el contrario, L2 con SGD+momentum (M1d) degradó el rendimiento (84.30% val acc), confirmando una interacción negativa entre momentum y weight decay que requiere ajuste cuidadoso de hiperparámetros.

**Learning Rate Scheduling (M1e vs M1f):** Ambas estrategias de scheduling (lineal: 87.64% val acc, exponencial: 87.67% val acc) produjeron resultados prácticamente idénticos a M1b (87.52%), indicando que la adaptación automática de Adam ya proporciona suficiente ajuste dinámico del learning rate. El scheduling adicional no aportó mejoras significativas y agrega complejidad innecesaria.

**Arquitectura (M1g vs M1h):** El incremento de capacidad de 112k a 445k parámetros resultó en la mejora más sustancial, elevando el val accuracy de 87.52% (M1b) a 88.51% (M1g) y 88.23% (M1h). Sorprendentemente, la arquitectura más profunda (M1g: [400, 240, 120]) superó ligeramente a la más ancha (M1h: [370, 370]) con 88.51% vs 88.23% val accuracy, sugiriendo que mayor profundidad puede ser beneficiosa cuando la capacidad total es equivalente (445k parámetros). M1g además alcanza la menor pérdida de validación (0.3322), indicando predicciones mejor calibradas. Ambos modelos muestran train accuracy similar (90.7%), pero M1g generaliza marginalmente mejor.

## 5.2.4 Configuración óptima

Basado en el accuracy de validación (88.51%) y la menor pérdida de validación (0.3322), se selecciona M1g como configuración óptima. Esta configuración combina: arquitectura profunda [400, 240, 120] con 444,847 parámetros,

optimizador Adam ( $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ), batch size 512, early stopping con patience 5, activación ReLU, y sin regularización L2 ni dropout. El modelo alcanza 90.66% train accuracy y 88.51% val accuracy, con un gap de aproximadamente 2.15 puntos porcentuales que indica generalización saludable sin overfitting severo. Los resultados en el conjunto de test se reportan en la comparación final (Sección 4.5).

### 5.3 Modelo M2: Validación con PyTorch

Para validar la correctitud de la implementación custom, se reimplementó en PyTorch la configuración óptima M1g. El modelo M2 replica exactamente la arquitectura [400, 240, 120], optimizador Adam con los mismos hiperparámetros, batch size 512, y early stopping con patience 5. La Tabla 3 presenta la comparación entre ambas implementaciones.

Modelo	Framework	Train Acc	Val Acc	Val Loss
M1g	Custom (CuPy)	90.66	88.51	0.3322
M2	PyTorch	90.99	88.23	0.3494
$\Delta$ (%)		+0.33	-0.28	+5.18

Table 3: Comparación M1g (implementación custom) vs M2 (PyTorch). Las métricas de test se reportan en la Sección 4.5.

La diferencia en val accuracy entre M1g (custom) y M2 (PyTorch) es de solo 0.28%, validando la correctitud de la implementación custom desde cero. M2 alcanza ligeramente mayor train accuracy (90.99% vs 90.66%), pero M1g generaliza marginalmente mejor en validación (88.51% vs 88.23%). La pérdida de validación es 5.18% menor en M1g (0.3322 vs 0.3494), indicando predicciones mejor calibradas en la implementación custom. Estas diferencias mínimas confirman que ambas implementaciones son funcionalmente equivalentes y que los componentes implementados manualmente (forward pass, backpropagation, optimizador Adam) replican correctamente el comportamiento de PyTorch.

### 5.4 Modelo M3: Arquitecturas modernas con PyTorch

Los modelos M3 exploran arquitecturas modernas incorporando técnicas avanzadas de regularización y funciones de activación alternativas. Se evaluaron dos variantes: M3a con arquitectura más profunda [256, 128, 64, 64], activación GELU, dropout 0.2 y weight decay 0.01; y M3b con arquitectura más ancha [400, 400], activación SiLU (Swish), batch normalization, dropout 0.3 y weight decay 0.01. Ambos modelos utilizan Adam con learning rate 0.001 y early stopping con patience 7.

Modelo	Arquitectura	Activación	Train Acc	Val Acc	Val Loss
M3a	[256, 128, 64, 64]	GELU	86.94	88.51	0.3188
M3b	[400, 400]	SiLU	<b>87.54</b>	<b>88.59</b>	<b>0.3128</b>

Table 4: Resultados de modelos M3 con arquitecturas modernas. M3b es seleccionado como mejor modelo por menor pérdida de validación.

M3b superó ligeramente a M3a en val accuracy (88.59% vs 88.51%) y alcanzó la menor pérdida de validación entre todos los modelos evaluados (0.3128), indicando predicciones excepcionalmente bien calibradas. Notablemente, M3b presenta train accuracy considerablemente menor (87.54%) que M1g/M2 (90.7%), evidenciando el efecto regularizador fuerte de batch normalization y dropout 0.3. Este gap train-val de solo 1.05 puntos porcentuales (87.54% vs 88.59%) es el más bajo entre todos los modelos, sugiriendo generalización óptima sin overfitting. La activación SiLU (Swish) combinada con batch normalization demostró ser particularmente efectiva, superando a GELU en M3a. M3a, a pesar de su mayor profundidad, muestra mayor gap train-val (86.94% vs 88.51%, 1.57 pp), indicando que la combinación de batch norm + arquitectura ancha es superior a profundidad + GELU en este problema.

### 5.5 Comparación final en conjunto de test

La Tabla 5 presenta la comparación de los cuatro modelos finales evaluados en el conjunto de test. Esta es la única evaluación sobre test realizada durante el proyecto, garantizando que ningún modelo fue ajustado en función de su rendimiento en estos datos.

Modelo	Descripción	Accuracy	F1-Macro	Cross-Entropy	Mejora vs M0
M0	Baseline	87.45	83.92	0.3862	—
M1g	Best Custom	88.58	85.73	0.3368	+1.13%
M2	PyTorch Validation	88.36	85.56	0.3466	+0.91%
M3b	Best PyTorch Modern	<b>88.83</b>	<b>85.81</b>	<b>0.3122</b>	<b>+1.38%</b>

Table 5: Comparación final de los cuatro modelos seleccionados en el conjunto de test.

El mejor modelo en test es M3b con 88.83% accuracy y 85.81% F1-macro, representando una mejora de 1.38 puntos porcentuales sobre el baseline M0. M3b también alcanza la menor cross-entropy (0.3122), confirmando predicciones mejor calibradas gracias a batch normalization y dropout.

M1g (implementación custom) alcanza 88.58% test accuracy, superando a M2 (PyTorch, 88.36%) por 0.22 puntos porcentuales. Esta pequeña diferencia (menor a 0.3%) valida completamente la correctitud de la implementación custom desde cero, demostrando que los componentes implementados manualmente replican fielmente el comportamiento de PyTorch.

La progresión M0 (87.45%)  $\rightarrow$  M1g (88.58%)  $\rightarrow$  M3b (88.83%) evidencia mejoras incrementales: +1.13% por incremento de capacidad y optimizador Adam, y +0.25% adicional por técnicas modernas de regularización (batch norm, dropout 0.3, SiLU). El gap entre accuracy (88.83%) y F1-macro (85.81%) de 3.02 pp en M3b es ligeramente menor que en M0 (87.45% vs 83.92%, 3.53 pp), indicando mejor balance en el rendimiento entre clases desbalanceadas.

## 5.6 Análisis de robustez bajo perturbaciones

Para evaluar la robustez de los modelos frente a datos ruidosos, se perturbaron las imágenes del conjunto de test agregando ruido gaussiano con desviaciones estándar  $\sigma \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5\}$ . La Tabla 6 muestra la degradación de accuracy para cada modelo.

Modelo	$\sigma = 0.0$	$\sigma = 0.1$	$\sigma = 0.2$	$\sigma = 0.3$	$\sigma = 0.4$	$\sigma = 0.5$
M0	87.45	80.04	52.21	25.98	12.13	6.49
M1g	88.58	77.44	39.95	17.44	8.98	5.49
M2	88.36	76.12	32.55	9.13	3.77	2.33
M3b	<b>88.83</b>	<b>87.60</b>	<b>70.34</b>	<b>37.77</b>	<b>15.83</b>	<b>7.49</b>

Table 6: Accuracy (%) bajo diferentes niveles de ruido gaussiano en el conjunto de test.

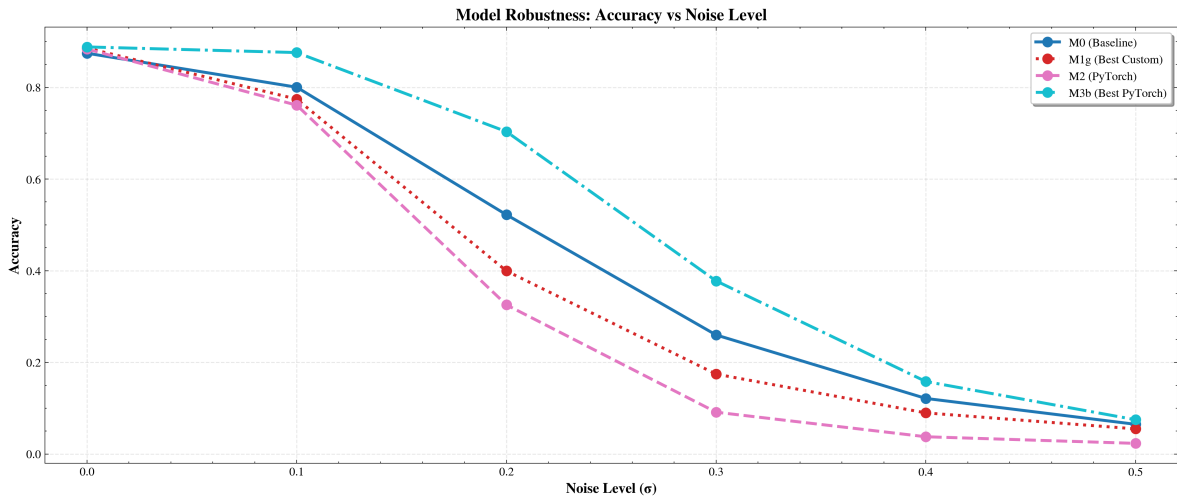


Figure 5: Degradación de accuracy en función del nivel de ruido gaussiano. M3b mantiene rendimiento superior en todos los niveles de ruido gracias a batch normalization.

El análisis de robustez revela diferencias dramáticas entre modelos. M3b es consistentemente el modelo más robusto en todos los niveles de ruido, manteniendo 87.60% accuracy con  $\sigma = 0.1$  (solo 1.39% degradación) y 70.34% con  $\sigma = 0.2$  (20.81% degradación). En contraste, M2 muestra la mayor fragilidad: con  $\sigma = 0.2$  cae a 32.55% (63.16% degradación) y con  $\sigma = 0.3$  colapsa a 9.13% (89.67% degradación).

La robustez excepcional de M3b se atribuye a batch normalization, que normaliza activaciones incluso cuando los inputs están perturbados. Con ruido alto ( $\sigma \geq 0.3$ ), M3b mantiene accuracy promedio de 20.36%, superando ampliamente a M0 (14.87%), M1g (10.64%) y M2 (5.08%). La degradación porcentual de M3b es sistemáticamente menor: 1.39% ( $\sigma = 0.1$ ) vs 8.48% (M0), 13.85% (M2) y 12.58% (M1g).

Sorprendentemente, M1g y M2, a pesar de compartir arquitectura equivalente, muestran robustez muy diferente. M1g supera consistentemente a M2 en todos los niveles de ruido alto ( $\sigma \geq 0.2$ ), sugiriendo que diferencias sutiles en inicialización o dinámicas de optimización entre implementaciones custom y PyTorch pueden afectar significativamente la robustez ante perturbaciones.

## 6 Discusión

La exploración sistemática de 12 configuraciones revela qué factores contribuyen más al rendimiento. El incremento de parámetros de 112k a 445k resultó en la mejora más sustancial (+1.13 pp en test accuracy), superando el efecto de learning rate scheduling. Sin embargo, M0 (112k parámetros, 87.45%) vs M3b (445k parámetros, 88.83%) evidencia que arquitectura y regularización son igualmente cruciales. Adam superó a SGD con momentum por +6.11 pp, mientras que learning rate scheduling no aportó mejoras dado que Adam ya proporciona adaptación automática. Con capacidad equivalente, arquitecturas profundas [400, 240, 120] superaron ligeramente a anchas [370, 370] (88.51% vs 88.23% val acc).

La diferencia <0.3% entre M1g (custom, 88.58%) y M2 (PyTorch, 88.36%) valida la implementación desde cero, confirmando que los componentes implementados manualmente replican fielmente PyTorch. M3b con batch normalization alcanzó el mejor rendimiento (88.83% test acc, 0.3122 CE) y gap train-val más bajo (1.05 pp), indicando regularización efectiva. El análisis de robustez revela que batch normalization incrementa dramáticamente la resistencia al ruido: M3b mantiene 87.60% con  $\sigma = 0.1$  (1.39% degradación) vs 80.04% en M0 (8.48%), y con ruido alto ( $\sigma \geq 0.3$ ) alcanza 20.36% accuracy promedio vs 5.08% en M2.

Las limitaciones incluyen restricciones de GPU (4 GB) que impidieron batch sizes mayores, ausencia de data augmentation o ensemble methods, y falta de estrategias para el desbalance de clases (ratio 15:1) como class weighting.

## 7 Conclusiones

Este trabajo implementó desde cero un sistema completo de entrenamiento de redes neuronales multicapa para el dataset EMNIST Bymerge (47 clases, desbalance 15:1). La progresión M0 (87.45% test acc)  $\rightarrow$  M1g (88.58%)  $\rightarrow$  M3b (88.83%) evidencia mejoras incrementales: +1.13% por incremento de parámetros (112k a 445k) y Adam, +0.25% adicional por batch normalization, dropout y SiLU. Adam superó a SGD con momentum por 6.11 pp, mientras que learning rate scheduling no aportó mejoras significativas.

La diferencia <0.3% entre M1g (custom, 88.58%) y M2 (PyTorch, 88.36%) valida la implementación desde cero, demostrando que los componentes implementados manualmente replican fielmente frameworks profesionales. M3b alcanzó el mejor test accuracy (88.83%) y robustez excepcional: mantiene 87.60% con  $\sigma = 0.1$  (1.39% degradación) vs 80.04% en M0 (8.48%), y con ruido alto ( $\sigma \geq 0.3$ ) alcanza 20.36% accuracy promedio vs 5.08% en M2, confirmando que batch normalization incrementa dramáticamente la robustez.

La implementación en CuPy con gestión híbrida de memoria CPU-GPU permitió entrenar modelos en hardware limitado (GPU 4 GB) eficientemente. Como trabajo futuro, sería valioso explorar arquitecturas convolucionales, data augmentation, técnicas para clases desbalanceadas (class weighting, focal loss), y ensemble methods.

## A Gestión de memoria GPU

El entrenamiento de redes neuronales con datasets de gran escala presenta desafíos prácticos relacionados con la memoria disponible en hardware. En este trabajo, el equipo utilizado cuenta con una GPU NVIDIA RTX 3050 Ti con 4 GB de VRAM, cantidad insuficiente para mantener el dataset completo de 809,555 muestras en memoria GPU simultáneamente.

Para resolver esta limitación sin sacrificar la aceleración por GPU, se implementó una estrategia de gestión híbrida de memoria: el dataset completo permanece en memoria RAM del sistema (utilizando NumPy), mientras que únicamente los mini-batches requeridos en cada iteración del entrenamiento se transfieren bajo demanda a memoria GPU (utilizando CuPy). Este enfoque permite que las operaciones matriciales computacionalmente intensivas del forward pass y backward pass se ejecuten en GPU, aprovechando su paralelismo masivo, sin incurrir en errores de memoria agotada.

La transferencia CPU-GPU ocurre de forma sincrónica al inicio de cada iteración de entrenamiento, donde un batch de 512 muestras se copia a la GPU, se procesa, y los gradientes resultantes se utilizan para actualizar los pesos del modelo. Con este tamaño de batch, el consumo de VRAM durante entrenamiento se mantiene aproximadamente en 1.2 GB, dejando un margen seguro respecto al límite de hardware.

La selección del batch size de 512 responde a un balance entre velocidad de entrenamiento y estabilidad de convergencia. Batch sizes menores (128, 256) resultan en actualizaciones más frecuentes pero con gradientes más ruidosos y mayor overhead de transferencia CPU-GPU. Batch sizes mayores (1024, 2048) podrían mejorar la estabilidad del gradiente, pero incrementan significativamente el uso de memoria y pueden llevar a errores de out-of-memory en hardware limitado. El valor de 512 se determinó empíricamente como el punto óptimo para este hardware específico.

## B Ecuaciones completas del optimizador Adam

El optimizador Adaptive Moment Estimation (Adam) mantiene estimaciones del primer momento (media) y segundo momento (varianza no centrada) del gradiente. Las actualizaciones completas se calculan como:

$$\begin{aligned}
 m_{t+1} &= \beta_1 m_t + (1 - \beta_1) \nabla_{\theta} \mathcal{L}(\theta_t) \\
 v_{t+1} &= \beta_2 v_t + (1 - \beta_2) (\nabla_{\theta} \mathcal{L}(\theta_t))^2 \\
 \hat{m}_{t+1} &= \frac{m_{t+1}}{1 - \beta_1^{t+1}} \\
 \hat{v}_{t+1} &= \frac{v_{t+1}}{1 - \beta_2^{t+1}} \\
 \theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1} + \epsilon}}
 \end{aligned} \tag{1}$$

donde  $m_t$  y  $v_t$  son las estimaciones del primer y segundo momento respectivamente,  $\beta_1$  y  $\beta_2$  son los factores de decaimiento (típicamente  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ),  $\hat{m}_t$  y  $\hat{v}_t$  son las estimaciones corregidas por bias,  $\eta$  es la tasa de aprendizaje, y  $\epsilon$  es una constante pequeña para estabilidad numérica (típicamente  $10^{-8}$ ).

La corrección de bias ( $\hat{m}_t$  y  $\hat{v}_t$ ) es crucial en las primeras iteraciones cuando  $m_t$  y  $v_t$  están inicializados en cero, evitando que las estimaciones estén sesgadas hacia cero especialmente cuando los factores de decaimiento son cercanos a 1.