Informe Técnico - Taller 2: Aprendizaje Automático

Nombre del Estudiante
Departamento de Física e Ingeniería
Universidad Nacional de Colombia

10 de diciembre de 2025

1. Introducción

Este informe presenta los resultados y análisis obtenidos del desarrollo del Taller 2. El trabajo se dividió en dos partes principales: la implementación robusta de la función de pérdida *Cross Entropy* y la construcción desde cero de una red neuronal utilizando un motor de diferenciación automática propio. Se realizaron diversos experimentos para entender el impacto de los hiperparámetros y la inicialización en el entrenamiento.

2. Análisis de Entropía Cruzada (Cross Entropy)

Se implementaron dos versiones de la función de pérdida *Cross Entropy*:

1. **Naive**: Implementación directa de la fórmula matemática.
2. **Optimized**: Implementación utilizando el truco *Log-Sum-Exp* para estabilidad numérica.

2.1. Resultados

- **Estabilidad Numérica**: La versión *Naive* falló (produciendo NaNs) cuando se probaron valores de entrada grandes (logits > 700), debido al desbordamiento (overflow) de la función exponencial. La versión Optimizada manejó estos casos correctamente, demostrando ser robusta ante valores extremos.
- **Rendimiento**: Ambas versiones mostraron tiempos de ejecución comparables para tamaños de *batch* pequeños y medianos. Sin embargo, la ligera sobrecarga computacional de la versión optimizada es insignificante comparada con el beneficio crítico de evitar inestabilidad numérica durante el entrenamiento.

3. Red Neuronal y Diferenciación Automática

Se construyó una librería de diferenciación automática (denominada "Autograd") que permite:

- Definir Tensores y operaciones matemáticas fundamentales (Suma, Multiplicación, MatMul, etc.).
- Calcular gradientes automáticamente mediante *backpropagation* (diferenciación en modo reverso).
- Construir redes neuronales modulares.

La arquitectura utilizada para los experimentos consistió en una red totalmente conectada (MLP) con 3 capas ocultas y funciones de activación ReLU, finalizando con una capa Softmax para la tarea de clasificación.

4. Análisis de Experimentos

4.1. Inicialización de Pesos

Se evaluaron diferentes estrategias de inicialización de pesos para observar su impacto crítico en la convergencia del modelo. Los resultados se visualizan en la Figura 1.

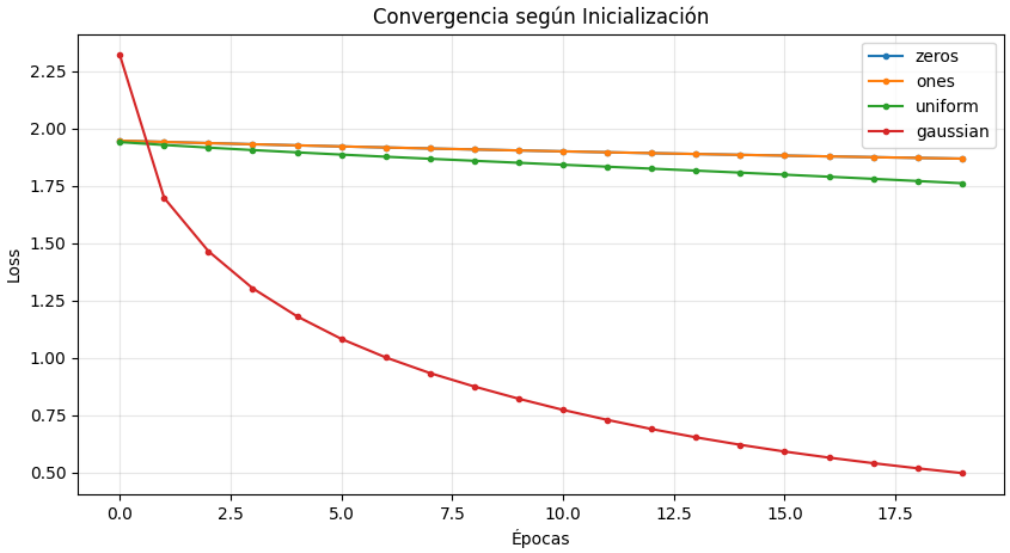

Figura 1: Comparación de la convergencia de la pérdida según el método de inicialización de pesos.

- **Zeros / Ones**: Como se evidencia en la gráfica (líneas azul y naranja que no descienden), estas estrategias resultaron en simetría de gradientes (todos los pesos de una capa se actualizan idénticamente) o gradientes nulos, impidiendo totalmente el aprendizaje efectivo del modelo.
- **Uniform**: Permitió el aprendizaje (línea verde), pero la convergencia fue notablemente más lenta que la estrategia óptima.

- **Gaussian (He Initialization)**: Esta estrategia (línea roja) mostró el mejor desempeño por un amplio margen. Al ajustar la varianza de los pesos iniciales según el tamaño de la capa anterior, facilitó una convergencia rápida y estable. Esto confirma la literatura sobre la importancia de la inicialización de He para redes profundas que utilizan activaciones ReLU.

4.2. Tamaño de Batch (Batch Size)

Se realizó un estudio exhaustivo del impacto del tamaño del *batch* tanto en la dinámica de convergencia como en la eficiencia computacional. Se probaron tamaños de 16, 32, 64, 128, 256, *Full Batch* y SGD (*Batch size* 1, ejecutado por 2 épocas). Los resultados se resumen en la Figura 2.

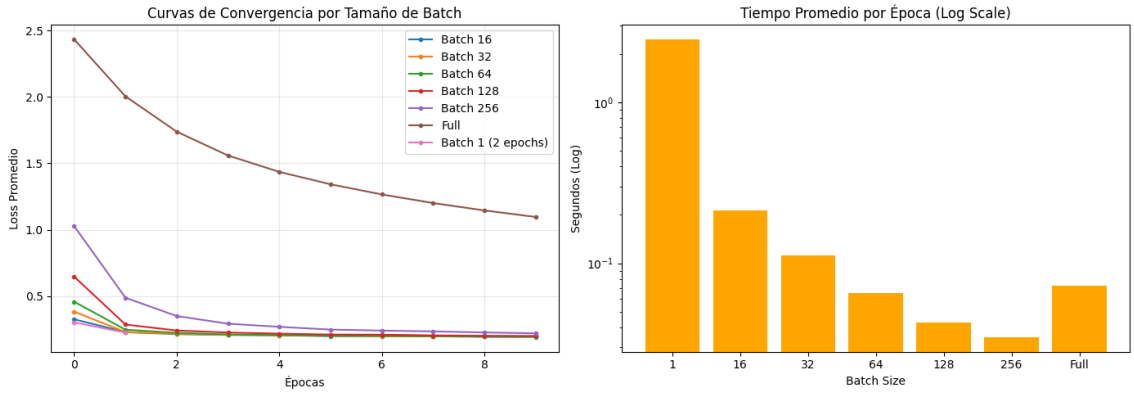

Figura 2: Izquierda: Curvas de convergencia de pérdida para distintos tamaños de batch. Derecha: Tiempo promedio de cómputo por época en escala logarítmica.

Análisis de Eficiencia Computacional: Como se observa en el panel derecho de la Figura 2 (escala logarítmica), el tiempo de cómputo por época es inversamente proporcional al tamaño del batch. Los tamaños de batch grandes (como Full Batch o 256) aprovechan eficientemente la vectorización de operaciones matriciales en NumPy y reducen el 'overhead' del bucle de entrenamiento en Python. Por el contrario, tamaños muy pequeños (como SGD o batch 16) sufren una penalización de tiempo significativa debido a la constante iteración del intérprete.

Análisis de Convergencia: El panel izquierdo muestra que el uso de mini-batches acelera drásticamente la convergencia en términos de épocas en comparación con el Batch Completo. Esto se debe a que la mayor frecuencia de actualizaciones de pesos (cientos de veces por época para batches pequeños) permite a la red aprender más rápido.

Mientras que la curva del Batch Completo es suave pero muy lenta, las curvas de los mini-batches presentan ruido"debido a la naturaleza estocástica de la estimación del gradiente. Sin embargo, esta inestabilidad es beneficiosa, pues permite al modelo escapar de mínimos locales y alcanzar valores de pérdida menores en menos tiempo.

Conclusión: Los resultados sugieren que tamaños de batch intermedios (e.g., 32 o 64) ofrecen el mejor compromiso entre la velocidad de aprendizaje (actualizaciones

frecuentes), la estabilidad del gradiente y la eficiencia computacional (aprovechamiento de la vectorización sin exceder la memoria).

4.3. Tasa de Aprendizaje (Learning Rate)

Se evaluó la sensibilidad del modelo ante distintas tasas de aprendizaje, como se muestra en la Figura 3.

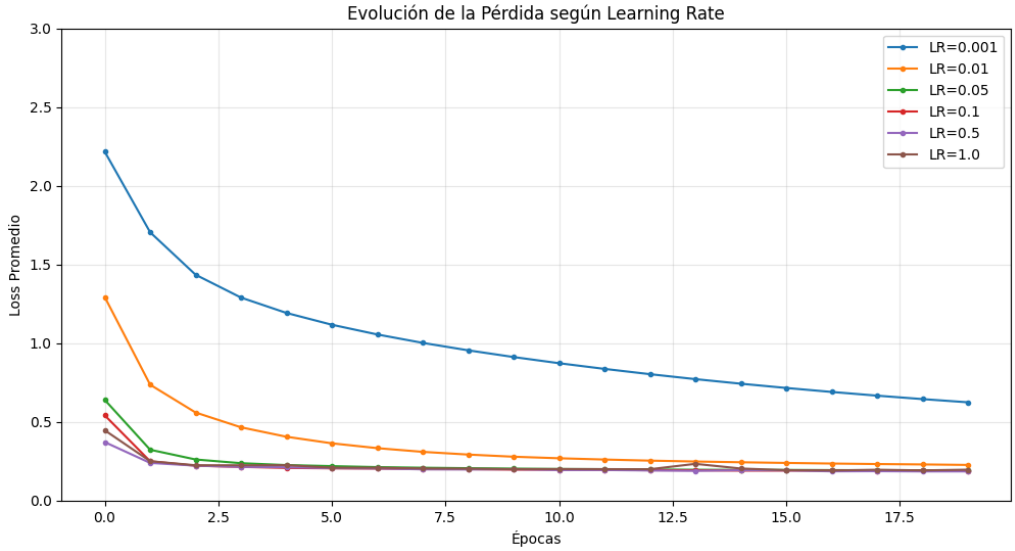

Figura 3: Evolución de la pérdida promedio según distintas tasas de aprendizaje (LR).

Se observa que tasas de aprendizaje muy bajas (LR=0.001) resultan en una convergencia excesivamente lenta. Para este conjunto de datos y arquitectura específicos, las tasas más altas (LR=0.5 y LR=1.0) lograron la convergencia más rápida y a valores de pérdida más bajos. Cabe destacar que, aunque efectivas aquí, tasas tan altas ($> 0{,}1$) suelen ser riesgosas en problemas más complejos, pudiendo causar divergencia o inestabilidad numérica si la superficie de error es muy irregular.

4.4. Uso de Momento (Momentum)

Se comparó el descenso de gradiente estándar ($\beta = 0{,}0$) contra versiones utilizando momento para acelerar el entrenamiento en las direcciones relevantes y amortiguar oscilaciones.

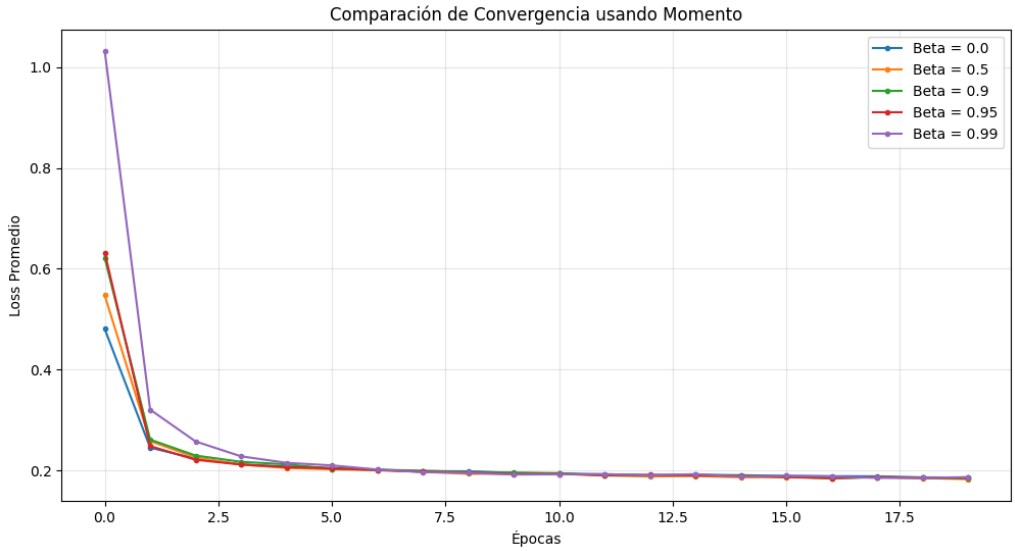

Figura 4: Comparación de convergencia variando el hiperparámetro Beta (momento).

La Figura 4 evidencia que la incorporación de momento mejora la velocidad de convergencia respecto a SGD puro (línea azul, $\beta = 0{,}0$). Un valor de $\beta = 0{,}9$ (línea verde) suele ser el estándar en la industria y demostró ser altamente efectivo aquí. Valores muy altos de inercia, como $\beta = 0{,}99$, pueden ser contraproducentes al introducir demasiada "pesadez.^en la actualización, aunque en este experimento no causaron divergencia.

5. Conclusiones Generales

1. **Estabilidad Numérica es Prioritaria**: Es un aspecto fundamental en la implementación de algoritmos de *Deep Learning*. El uso de trucos como Log-Sum-Exp en la entropía cruzada es obligatorio para evitar fallos catastróficos (NaNs) en entornos reales.
2. **La Inicialización no es Trivial**: Una mala inicialización (como ceros o unos) rompe la capacidad de aprendizaje de la red desde el inicio. Métodos adecuados como *He Initialization* son necesarios para romper la simetría y mantener el flujo de gradientes en redes profundas con ReLU.
3. **El Compromiso del Batch Size**: El tamaño del batch es un hiperparámetro crítico que balancea la calidad de la estimación del gradiente contra la velocidad de cómputo. Los mini-batches intermedios demostraron ser superiores tanto a SGD puro (muy ruidoso y lento computacionalmente) como al Full Batch (muy lento en convergencia).