Universidad de Alicante

Ingeniería Informática

Sistemas Inteligentes

Práctica 2:

Visión artificial y aprendizaje

Autor:

Jaime Hernández Delgado (jhd3)

Curso 2024/2025

Índice

1. Introducción

1.1. Objetivos

Esta práctica tiene como objetivo principal el desarrollo de capacidades en el ámbito del aprendizaje automático y la visión artificial, centrándose específicamente en la implementación y optimización de redes neuronales para la clasificación de imágenes. Los objetivos específicos son:

- Comprender en profundidad el funcionamiento de las redes neuronales artificiales como técnicas de aprendizaje supervisado, con especial énfasis en:
  - Perceptrones multicapa (MLP)
  - Redes neuronales convolucionales (CNN)
- Adquirir conocimientos fundamentales sobre el procesamiento de imágenes:
  - Entender el concepto de imagen digital y píxel
  - Comprender cómo las técnicas de aprendizaje supervisado pueden aplicarse para la clasificación de imágenes
- Dominar el algoritmo de retropropagación (backpropagation) y sus componentes esenciales:
  - Funciones de activación
  - Optimizadores
  - Funciones de pérdida
  - Métricas de evaluación
- Desarrollar habilidades prácticas en el uso de Keras:
  - Implementación eficiente de redes neuronales
  - Configuración y ajuste de hiperparámetros
  - Evaluación de modelos
- Aprender a optimizar el rendimiento de redes neuronales mediante:
  - Ajuste del número de épocas de entrenamiento
  - Configuración del tamaño de batch
  - Selección de funciones de activación apropiadas
  - Diseño de arquitecturas de red efectivas
- Desarrollar competencias en visualización y análisis de resultados:
  - Utilización de Matplotlib para la generación de gráficas
  - Interpretación de métricas de rendimiento
  - Análisis de matrices de confusión

- Adquirir experiencia en la automatización de procesos:
    - Desarrollo de scripts para pruebas automáticas
    - Generación automatizada de resultados
    - Creación eficiente de contenido para el informe

A través de estos objetivos, se busca desarrollar una comprensión integral de las redes neuronales y su aplicación práctica en problemas de visión artificial, desde la implementación básica hasta la optimización avanzada de modelos.

1.2. Herramientas y tecnologías utilizadas

Para el desarrollo de esta práctica se han utilizado diversas herramientas y tecnologías del ecosistema de Python orientadas al aprendizaje automático y la ciencia de datos:

1.2.1. Bibliotecaa principal

- **Keras**: API de alto nivel que se ejecuta sobre TensorFlow, que ofrece:
    - Interfaz intuitiva para la construcción de redes neuronales
    - Implementación rápida de modelos complejos
    - Amplia variedad de capas predefinidas y funciones de activación
    - Herramientas de preprocesamiento de datos

1.2.2. Bibliotecas de apoyo

- **NumPy**: Biblioteca fundamental para la computación científica en Python:
    - Manipulación eficiente de arrays multidimensionales
    - Operaciones matemáticas vectorizadas
    - Funciones para el preprocesamiento de datos
- **Matplotlib**: Biblioteca para la creación de visualizaciones:
    - Generación de gráficas de evolución del entrenamiento
    - Visualización de imágenes del dataset
    - Creación de gráficas comparativas de resultados

1.2.3. Dataset

- **CIFAR-10**: Conjunto de datos estándar que contiene:
    - 60.000 imágenes en color de 32x32 píxeles
    - 10 clases diferentes (avión, automóvil, pájaro, gato, ciervo, perro, rana, caballo, barco y camión)
    - 50.000 imágenes para entrenamiento y 10.000 para pruebas
    - Distribución equilibrada de clases

1.2.4. Control de versiones y documentación

- **LaTeX**: Sistema de composición de textos utilizado para la generación de esta memoria
- **Git**: Sistema de control de versiones para el seguimiento de cambios en el código

2. Desarrollo

2.1. Preprocesamiento de datos

El preprocesamiento de los datos es un paso fundamental para el correcto funcionamiento de las redes neuronales. En esta práctica, trabajamos con el dataset CIFAR10, que requiere cierta preparación antes de poder ser utilizado efectivamente por nuestros modelos.

2.1.1. Carga del dataset

El primer paso consiste en cargar el dataset CIFAR10 utilizando las utilidades proporcionadas por Keras:

```
(X_train, Y_train), (X_test, Y_test) = keras.datasets.cifar10.load_data
    ()
```

Tras la carga inicial, los datos presentan las siguientes características:

- **X_train**: Array de 50.000 imágenes de entrenamiento de dimensiones (32, 32, 3)
- **Y_train**: Array de 50.000 etiquetas de entrenamiento
- **X_test**: Array de 10.000 imágenes de prueba de dimensiones (32, 32, 3)
- **Y_test**: Array de 10.000 etiquetas de prueba

2.1.2. Normalización de las imágenes

Las imágenes originales tienen valores de píxeles en el rango [0, 255]. Para mejorar el entrenamiento de las redes neuronales, es necesario normalizar estos valores al rango [0, 1]:

```
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

Esta normalización es importante por varios motivos:

- Evita problemas de escala en los cálculos numéricos
- Ayuda a que el proceso de entrenamiento sea más estable
- Facilita la convergencia del algoritmo de optimización

2.1.3. Preprocesamiento específico para MLP

Para las redes neuronales tipo MLP, es necesario aplanar las imágenes ya que estas redes esperan datos de entrada unidimensionales:

```
X_train_mlp = X_train.reshape((X_train.shape[0], -1))  # (50000, 3072)
X_test_mlp = X_test.reshape((X_test.shape[0], -1))     # (10000, 3072)
```

Esta transformación convierte cada imagen de una matriz 32x32x3 en un vector de 3.072 elementos (32 * 32 * 3).

2.1.4. Codificación de etiquetas

Las etiquetas originales están en formato de índice (números del 0 al 9). Para el entrenamiento, necesitamos convertirlas a formato one-hot encoding:

```
Y_train = keras.utils.to_categorical(Y_train, 10)
Y_test = keras.utils.to_categorical(Y_test, 10)
```

Esta transformación convierte cada etiqueta en un vector de 10 elementos donde:

- El valor 1 indica la clase correcta
- El resto de valores son 0
- Por ejemplo, la clase 3 se convierte en [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

2.1.5. Consideraciones para CNN

Para las redes neuronales convolucionales (CNN), no es necesario aplanar las imágenes, ya que estas redes están diseñadas para trabajar directamente con datos bidimensionales. Por lo tanto, mantendremos la estructura original de las imágenes (32, 32, 3) cuando trabajemos con CNNs.

2.1.6. Validación de los datos

Después del preprocesamiento, es importante verificar:

- Que los valores de los píxeles estén en el rango [0, 1]
- Que las dimensiones de los arrays sean correctas
- Que las etiquetas estén correctamente codificadas

```
print(f"Rango de valores X_train: [{X_train.min()}, {X_train.max()}]")
print(f"Forma X_train MLP: {X_train_mlp.shape}")
print(f"Forma Y_train: {Y_train.shape}")
```

Este preprocesamiento es crucial para el correcto funcionamiento de nuestros modelos y nos permitirá obtener mejores resultados durante el entrenamiento.

2.2. Tarea A: Implementación básica de MLP

2.2.1. Fundamentos teóricos

El Perceptrón Multicapa (Multi-Layer Perceptron, MLP) es una de las arquitecturas más fundamentales de redes neuronales artificiales. Se compone de múltiples capas de neuronas artificiales organizadas de manera jerárquica, donde cada neurona en una capa está conectada con todas las neuronas de la capa siguiente.

Estructura básica: Un MLP típico consta de tres tipos principales de capas:

- **Capa de entrada**: Recibe los datos de entrada (en nuestro caso, los píxeles de la imagen)
- **Capas ocultas**: Realizan transformaciones no lineales de los datos
- **Capa de salida**: Produce la predicción final (en nuestro caso, la clasificación de la imagen)

Funcionamiento: El proceso de una neurona artificial se puede describir matemáticamente como:

$$y = f(\sum_{i=1}^{n} w_i x_i + b)$$

Donde:

- x_i son las entradas
- w_i son los pesos asociados a cada entrada
- b es el sesgo (bias)
- f es la función de activación
- y es la salida de la neurona

Función de activación: En nuestra implementación inicial utilizamos la función sigmoid para las capas ocultas:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Y softmax para la capa de salida:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Proceso de aprendizaje: El aprendizaje en un MLP ocurre mediante el algoritmo de retropropagación (backpropagation):

1. **Propagación hacia adelante**: Los datos atraviesan la red, generando una predicción
2. **Cálculo del error**: Se compara la predicción con el valor real

3. **Retropropagación**: El error se propaga hacia atrás en la red

4. **Actualización de pesos**: Se ajustan los pesos para minimizar el error

2.3. Tarea B: Ajuste del número de épocas

2.3.1. Fundamentos teóricos

En el contexto del aprendizaje automático, una época representa una pasada completa a través de todo el conjunto de datos de entrenamiento. El número de épocas es un hiperparámetro crucial que determina cuántas veces el algoritmo procesará el conjunto de datos completo durante el entrenamiento.

El sobreentrenamiento (overfitting) es un fenómeno que ocurre cuando un modelo aprende con demasiado detalle los datos de entrenamiento, hasta el punto de memorizar sus particularidades y ruido, en lugar de aprender patrones generales. Esto resulta en:

- Un rendimiento excelente en los datos de entrenamiento
- Un rendimiento deficiente en datos nuevos (baja capacidad de generalización)
- Una brecha creciente entre el rendimiento en entrenamiento y validación

La relación entre el número de épocas y el sobreentrenamiento es directa:

- Pocas épocas pueden resultar en subentrenamiento (underfitting), donde el modelo no ha aprendido suficientemente los patrones de los datos
- Demasiadas épocas pueden llevar al sobreentrenamiento, donde el modelo memoriza los datos de entrenamiento
- El desafío está en encontrar el punto óptimo entre estos dos extremos

2.3.2. Experimentación

Para encontrar el número óptimo de épocas, se realizaron experimentos con diferentes configuraciones:

Configuración del experimento:

- Valores de épocas probados: [5, 10, 20, 50, 100]
- Arquitectura del modelo: MLP con una capa oculta de 32 neuronas
- Función de activación: sigmoid en la capa oculta, softmax en la capa de salida
- Optimizador: Adam
- División de datos: 90 % entrenamiento, 10 % validación

Métricas monitorizadas:

- Precisión (accuracy) en entrenamiento y validación
- Tiempo de entrenamiento
- Pérdida (loss) en entrenamiento y validación

2.3.3. Resultados y análisis

Análisis de la precisión: La precisión del modelo evoluciona de la siguiente manera:

- 5 épocas: 0.39 (insuficiente entrenamiento)
- 20 épocas: 0.43 (mejora significativa)
- 50 épocas: 0.453 (mejor rendimiento)
- 100 épocas: 0.44 (ligera degradación)

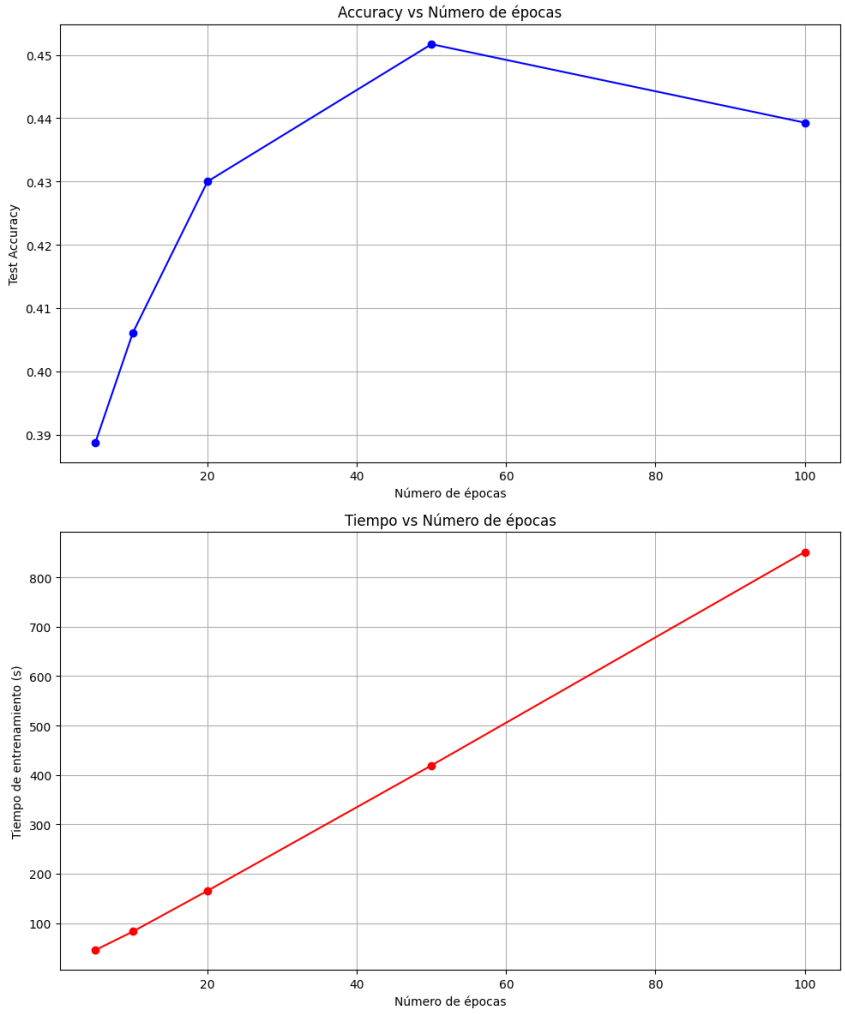

Figura 1: Evolución de la precisión según el número de épocas

Análisis del coste computacional: El tiempo de entrenamiento muestra una relación lineal con el número de épocas:

- 5 épocas: ≈ 50 segundos
- 50 épocas: ≈ 400 segundos
- 100 épocas: ≈ 800 segundos

Análisis de las curvas de aprendizaje: Las curvas de aprendizaje del mejor modelo (50 épocas) revelan:

- Una mejora constante en la precisión de entrenamiento hasta alcanzar 0.50
- Una estabilización de la precisión de validación alrededor de 0.43-0.45
- Una brecha creciente entre entrenamiento y validación, indicando inicio de sobreentrenamiento

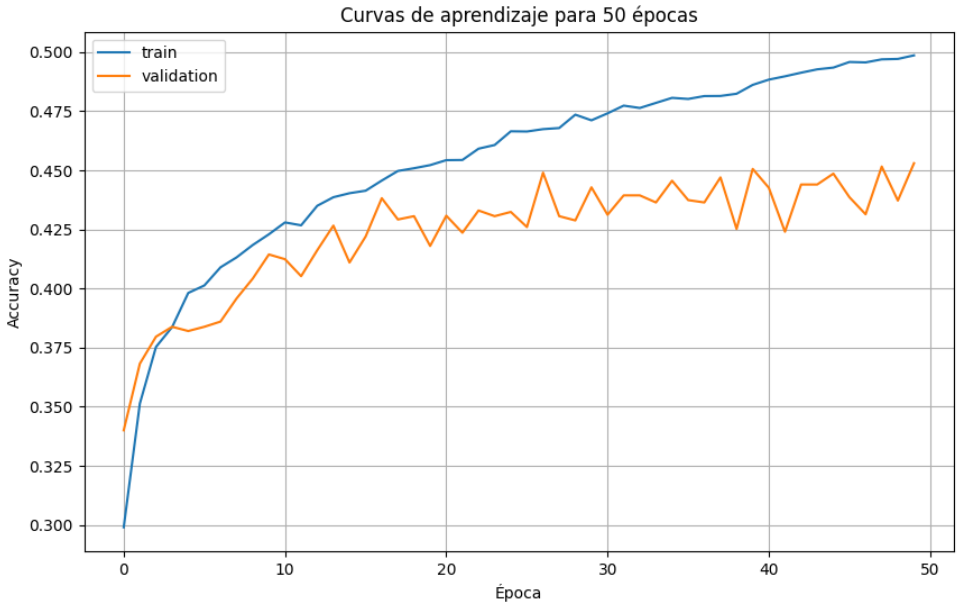

Figura 2: Curvas de aprendizaje para el modelo de 50 épocas

Conclusiones:

- El número óptimo de épocas para este modelo es 50, proporcionando el mejor balance entre precisión y tiempo de entrenamiento
- Entrenar más allá de 50 épocas resulta en:
  - Mayor tiempo de computación
  - Inicio de sobreentrenamiento
  - Degradación de la capacidad de generalización
- Se observa una clara relación entre el número de épocas y el riesgo de sobreentrenamiento

2.4. Tarea C: Ajuste del tamaño de batch

2.4.1. Fundamentos teóricos

El tamaño de batch (batch size) es un hiperparámetro que define cuántas muestras procesa el modelo antes de actualizar sus pesos. Este parámetro tiene un impacto significativo tanto en el proceso de aprendizaje como en el rendimiento computacional del modelo.

Aspectos clave del batch size:

- **Impacto en el aprendizaje**:
  - Afecta la estimación del gradiente
  - Influye en la capacidad de generalización
  - Determina la estabilidad del entrenamiento
- **Consideraciones computacionales**:
  - Define el uso de memoria durante el entrenamiento
  - Afecta la velocidad de convergencia
  - Impacta en el aprovechamiento del hardware

2.4.2. Experimentación

Para encontrar el tamaño de batch óptimo, se realizaron experimentos con las siguientes configuraciones:

Configuración del experimento:

- Valores de batch size: [16, 32, 64, 128, 256]
- Número fijo de épocas: 50
- Arquitectura: MLP con 32 neuronas en la capa oculta
- Función de activación: sigmoid
- División de datos: 90 % entrenamiento, 10 % validación

2.4.3. Resultados y análisis

Análisis del rendimiento: Los resultados muestran una clara relación entre el tamaño del batch y el rendimiento del modelo:

- **Batch size 16**:
  - Accuracy: 0.418

    - Tiempo: 775 segundos
    - Mayor variabilidad en el entrenamiento
- **Batch size 64**:
    - Accuracy: 0.449
    - Tiempo: 420 segundos
    - Mejor balance precisión/tiempo
- **Batch size 256**:
    - Accuracy: 0.457
    - Tiempo: 140 segundos
    - Mayor eficiencia computacional

Análisis del coste computacional: Se observa una clara relación inversa entre el tamaño del batch y el tiempo de entrenamiento:

- Reducción significativa del tiempo al aumentar el batch size
- Mejora de eficiencia más pronunciada entre 16 y 128
- Saturación de la mejora en tiempos a partir de 128

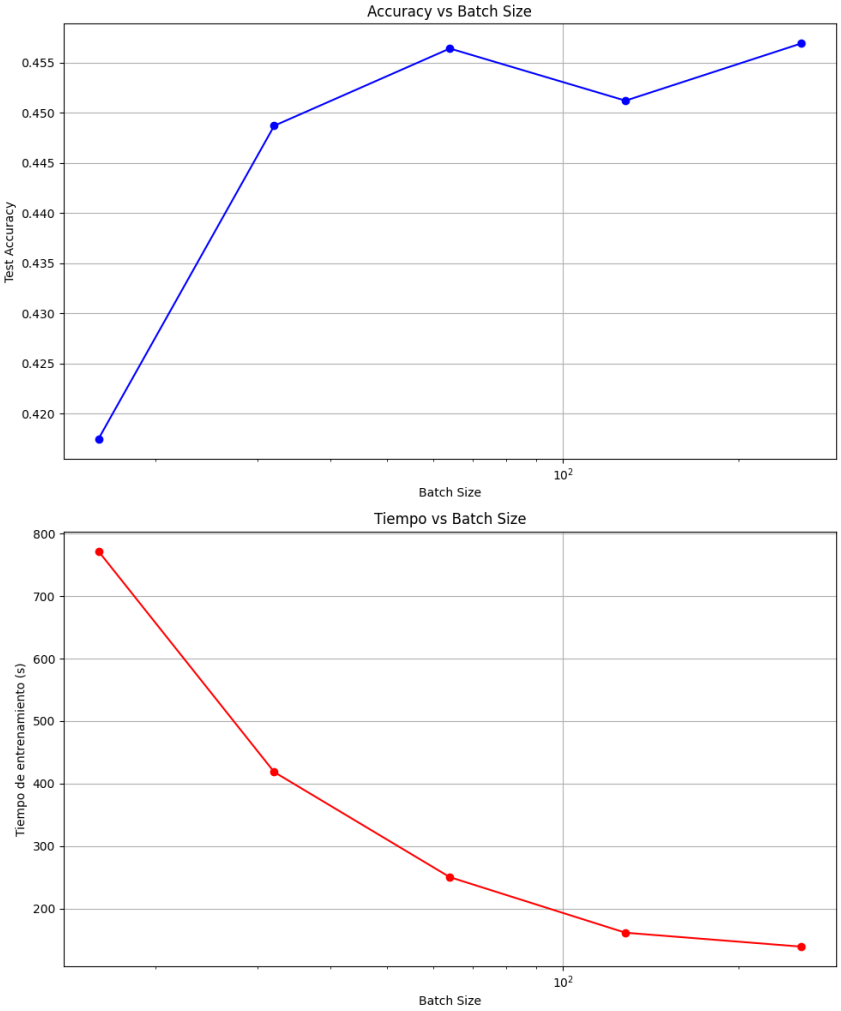

Figura 3: Tiempo de entrenamiento según el tamaño de batch

Análisis de las curvas de aprendizaje: Para el batch size de 256 (mejor rendimiento):

- Convergencia estable y consistente
- Diferencia moderada entre train y validation
- Mejora continua sin signos claros de sobreentrenamiento

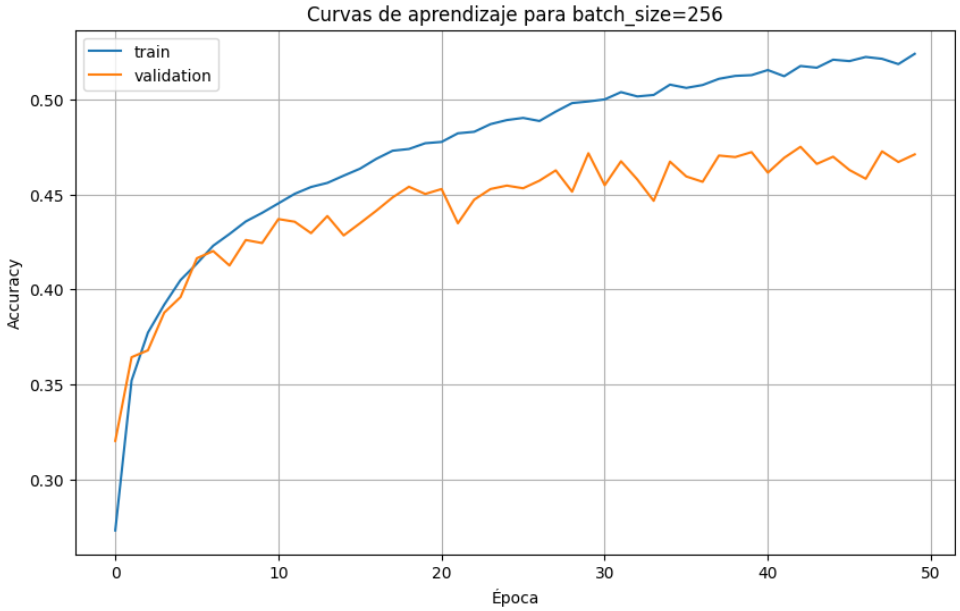

Figura 4: Curvas de aprendizaje para batch size=256

Conclusiones:

- El tamaño de batch óptimo es 256, proporcionando:
  - Mejor accuracy (0.457)
  - Tiempo de entrenamiento reducido (140s)
  - Estabilidad en el aprendizaje
- Se observa que:
  - Batches pequeños resultan en entrenamiento más lento
  - Batches grandes mejoran la eficiencia sin comprometer el rendimiento
  - Existe un punto de equilibrio entre precisión y tiempo

2.5. Tarea D: Funciones de activación

2.5.1. Fundamentos teóricos

Las funciones de activación son componentes críticos en las redes neuronales que introducen no linealidad en el modelo. Cada función tiene características específicas:

- **Sigmoid**: Rango [0,1], tradicionalmente usada pero propensa a desvanecimiento del gradiente
- **ReLU**: $f(x) = max(0, x)$, reduce el problema del desvanecimiento del gradiente
- **Tanh**: Similar a sigmoid pero con rango [-1,1]
- **ELU**: Variante de ReLU que puede manejar valores negativos

2.5.2. Experimentación

Se realizaron pruebas manteniendo la arquitectura base del MLP (32 neuronas en la capa oculta) y modificando únicamente la función de activación. Parámetros:

- Épocas: 50
- Batch size: 256
- Optimizador: Adam

2.5.3. Resultados y análisis

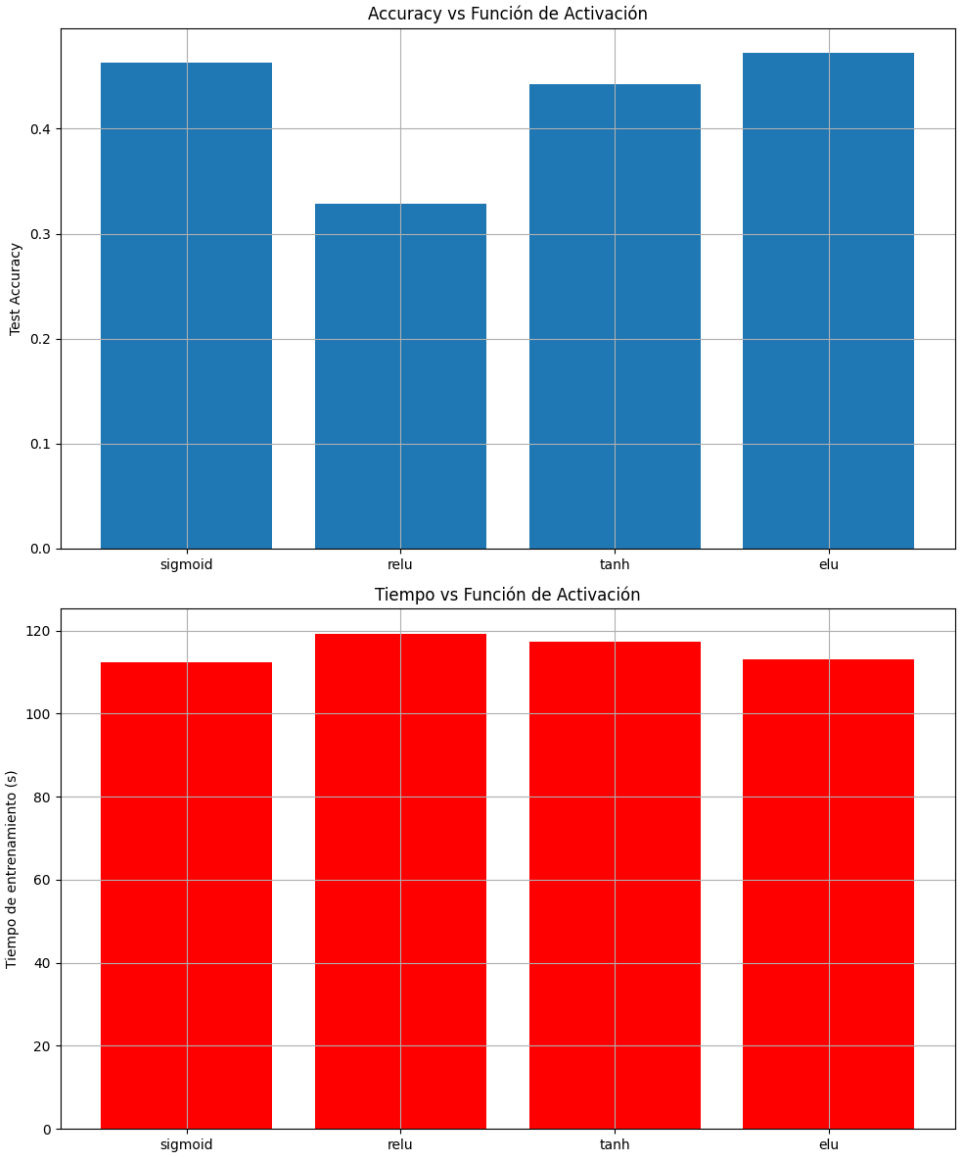

Figura 5: Comparación de funciones de activación

Las funciones sigmoid y elu mostraron el mejor rendimiento (accuracy 0.45), mientras que relu tuvo el peor desempeño. Los tiempos de entrenamiento fueron similares para todas las funciones, con diferencias menores al 10

2.6. Tarea E: Ajuste del número de neuronas

2.6.1. Fundamentos teóricos

El número de neuronas en la capa oculta determina la capacidad del modelo para aprender patrones complejos:

- Pocas neuronas: Underfitting (capacidad insuficiente)
- Demasiadas neuronas: Posible overfitting y mayor coste computacional

2.6.2. Experimentación

Se evaluaron diferentes configuraciones con [16, 32, 64, 128, 256] neuronas en la capa oculta, manteniendo:

- Función de activación: ELU (mejor resultado de tarea D)
- Épocas: 50
- Batch size: 256

2.6.3. Resultados y análisis

Los resultados muestran una mejora significativa en accuracy al aumentar el número de neuronas hasta 128, donde se alcanza un accuracy de aproximadamente 0.48. Sin embargo, el tiempo de entrenamiento aumenta exponencialmente con el número de neuronas. La configuración de 128 neuronas ofrece el mejor balance entre rendimiento y coste computacional.

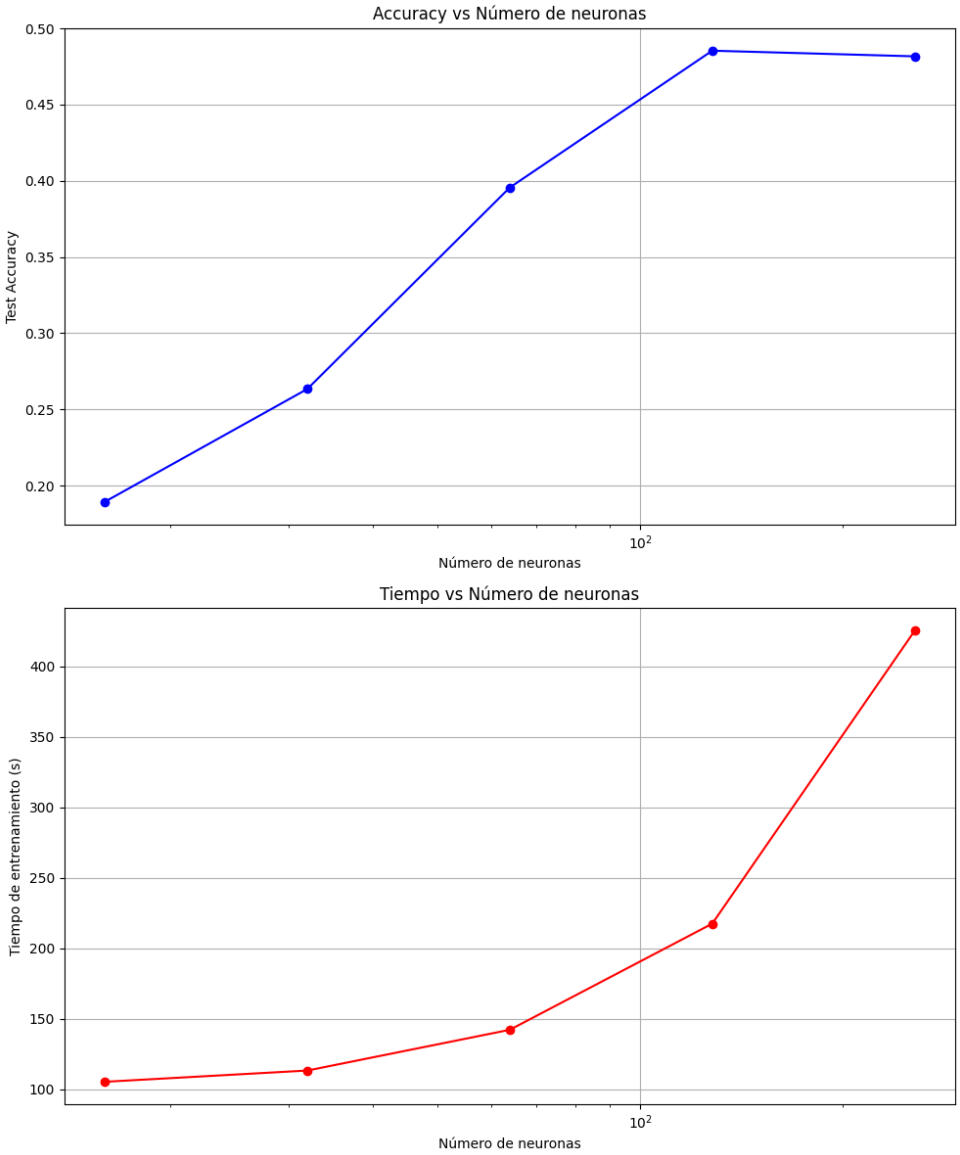

Figura 6: Número de neuronas

2.7. Tarea F: Optimización de MLP multicapa

2.7.1. Fundamentos teóricos

Las arquitecturas multicapa permiten aprender representaciones jerárquicas de los datos, donde cada capa puede especializarse en diferentes niveles de abstracción. El diseño de la arquitectura debe considerar:

- Profundidad (número de capas)
- Anchura (neuronas por capa)
- Patrón de conectividad entre capas

2.7.2. Experimentación

Se probaron diferentes arquitecturas:

- Una capa: [64]
- Dos capas: [32→32], [64→32], [32→64]
- Tres capas: [128→64→32]

2.7.3. Resultados y análisis

La arquitectura [128→64→32] mostró el mejor rendimiento (accuracy 0.50), con un tiempo de entrenamiento aceptable. Esta configuración permite una reducción gradual de dimensionalidad que parece beneficiar el aprendizaje. Las curvas de aprendizaje muestran una convergencia estable con una brecha moderada entre entrenamiento y validación, sugiriendo un buen balance entre capacidad de ajuste y generalización.

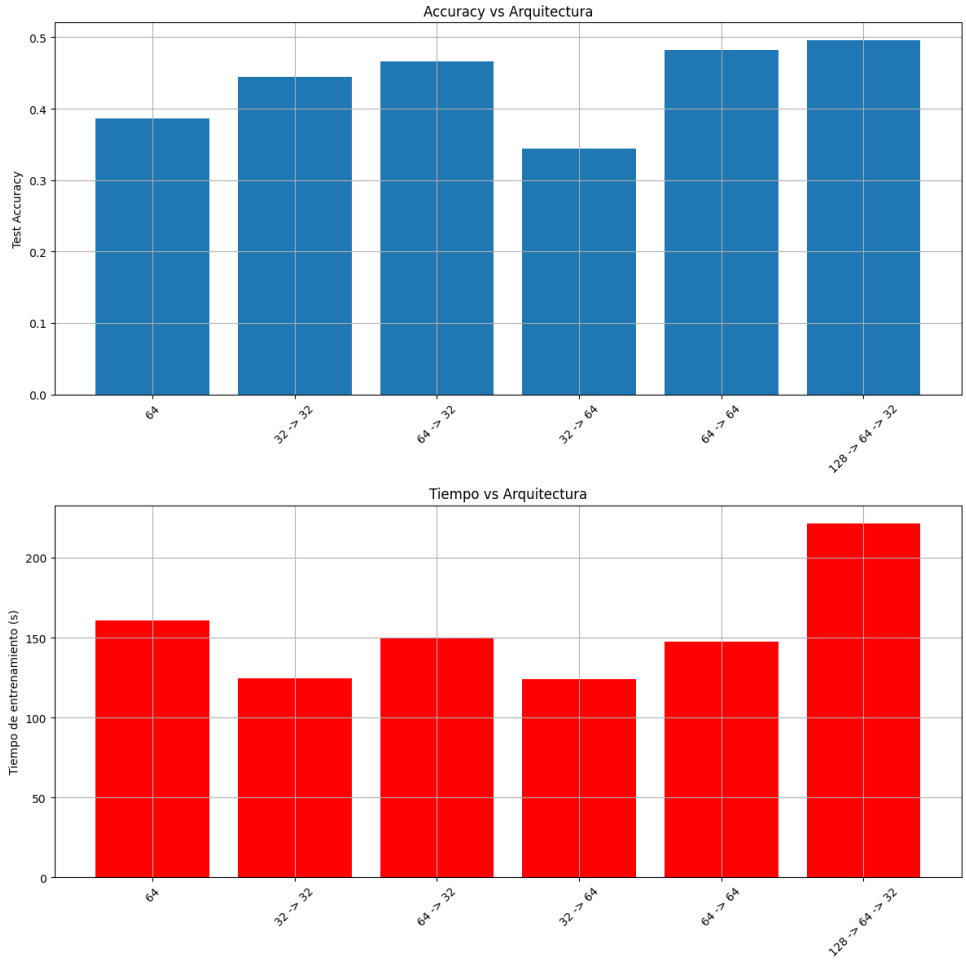

Figura 7: Arquitectura [128→64→32]

2.8. Tarea G: Implementación de CNN

2.8.1. Fundamentos teóricos

Las CNN se distinguen de otras redes neuronales por su mejor desempeño con entradas de señal de imagen, voz o audio. Tienen tres tipos principales de capas, que son:

1. Capa convolucional

2. Capa de agrupamiento

3. Capa totalmente conectada (FC)

La capa convolucional es la primera capa de una red convolucional. Con cada capa, la CNN aumenta su complejidad, identificando mayores porciones de la imagen. Las primeras capas se enfocan en características simples, como colores y bordes. A medida que los datos de la imagen avanzan a través de las capas de CNN, se comienzan a reconocer elementos o formas más grandes del objeto, hasta que finalmente se identifica el objeto previsto.

2.8.2. Resultados y análisis

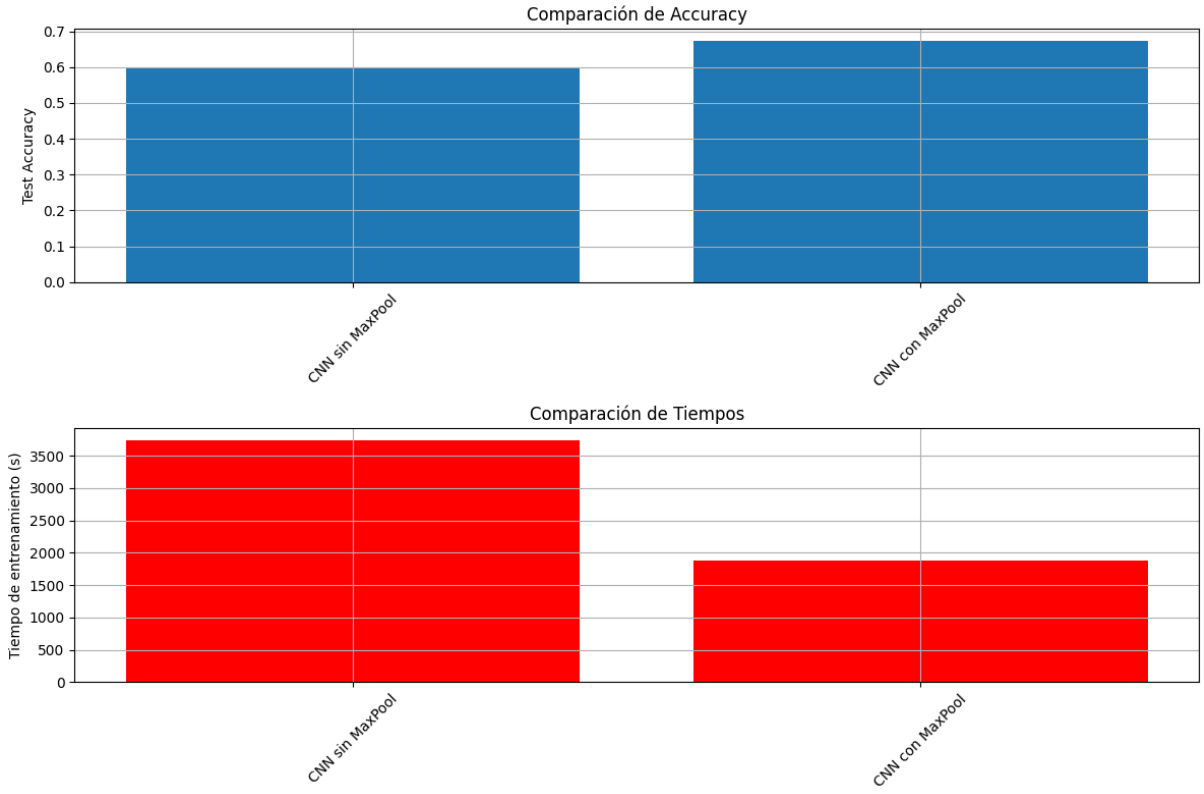

Figura 8: Enter Caption

2.9. Tarea H: Ajuste del tamaño de kernel

La relación entre el tamaño de los filtros y el tamaño de los datos de entrada es crucial. Si los datos de entrada son grandes, los filtros más grandes pueden ser más efectivos para capturar características globales en un solo paso. Sin embargo, si los datos de entrada son más pequeños, los filtros grandes podrían capturar demasiada información a una escala no relevante y pueden perder detalles importantes.

El tamaño del filtro debe ajustarse en función del tamaño de las imágenes de entrada. Para entradas más grandes, los filtros mayores pueden ser útiles para una mayor captura de información con menos capas, mientras que para entradas pequeñas, los filtros más pequeños permiten una mayor flexibilidad y capacidad de generalización.

Hipótesis a Probar

Redes con Filtros Pequeños (3x3, 5x5): Se espera que estos filtros aprendan características finas pero requieran más capas Conv2D para aprender representaciones jerárquicas complejas. Redes con Filtros Grandes (7x7, 9x9): Estas redes deberían ser capaces de aprender características grandes con menos capas, pero su costo computacional será mayor. MaxPooling2D: Es posible que su utilidad dependa del tamaño de los filtros. Redes con filtros grandes pueden no necesitar tanto MaxPooling2D como aquellas con filtros pequeños.

Al ajustar el tamaño del filtro en las capas convolucionales, se debe considerar un balance entre la capacidad de la red para detectar características y el costo computacional. Las redes con filtros grandes son más eficientes en términos de detección de características grandes pero pueden ser más costosas computacionalmente. Por otro lado, las redes con filtros pequeños pueden requerir más capas y mayor tiempo de entrenamiento para aprender representaciones complejas, pero su costo es más manejable. La elección de usar o no MaxPooling2D dependerá de la arquitectura específica y del tamaño de los filtros utilizados.

2.10. Tarea I: Optimización de la arquitectura CNN

2.11. Capa Convolucional (Conv2D)

Las capas convolucionales son el núcleo de las CNN, encargadas de aplicar filtros a las imágenes para extraer características locales. Cuantas más capas convolucionales haya, más abstractas y complejas serán las características aprendidas. Sin embargo, un número mayor de capas puede llevar a un sobreajuste si la red no está debidamente regularizada.

2.12. Gráfica de Resultados

La gráfica de resultados debe mostrar la relación entre el *tiempo de entrenamiento* y la *precisión en el conjunto de prueba*. A continuación, se espera que la gráfica tenga la siguiente forma:

- En el eje X, se representará el tiempo de entrenamiento en segundos.
- En el eje Y, se representará la precisión en el conjunto de prueba.

Se anticipa que, a medida que la red sea más compleja, tanto la precisión como el tiempo de entrenamiento aumentarán, pero la relación no es necesariamente lineal, debido al sobreajuste y la capacidad de generalización de la red.

Figura 9: Relación entre el tiempo de entrenamiento y la precisión en el conjunto de prueba para diferentes arquitecturas CNN.

3. Conclusiones

4. Bibliografía

Referencias

[1] Chollet, François et al., *Keras Documentation*, 2015, https://keras.io/

[2] Chollet, François et al., *Keras Layers API*, 2015, https://keras.io/api/layers/

[3] Chollet, François et al., *Keras Activations API*, 2015, https://keras.io/api/layers/activations/

[4] Krizhevsky, Alex, *Learning Multiple Layers of Features from Tiny Images*, 2009, Technical Report, University of Toronto

[5] LeCun, Yann and Bengio, Yoshua and Hinton, Geoffrey, *Deep Learning*, Nature, 2015, vol. 521, no. 7553, pp. 436-444

[6] Goodfellow, Ian and Bengio, Yoshua and Courville, Aaron, *Deep Learning*, MIT Press, 2016, http://www.deeplearningbook.org

[7] He, Kaiming et al., *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, 2015, Proceedings of the IEEE International Conference on Computer Vision

[8] Ioffe, Sergey and Szegedy, Christian, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015, International Conference on Machine Learning

[9] Kingma, Diederik P. and Ba, Jimmy, *Adam: A Method for Stochastic Optimization*, 2014, arXiv preprint arXiv:1412.6980

[10] Zeiler, Matthew D. and Fergus, Rob, *Visualizing and Understanding Convolutional Networks*, 2014, European Conference on Computer Vision

[11] Simonyan, Karen and Zisserman, Andrew, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014, arXiv preprint arXiv:1409.1556

[12] Srivastava, Nitish et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014, Journal of Machine Learning Research

[13] Chollet, François, *Building Powerful Image Classification Models Using Very Little Data*, The Keras Blog, https://keras.io/examples/vision/

[14] Chollet, François, *Hyperparameter Tuning*, The Keras Blog, https://keras.io/guides/keras_tuner/

A. Anexos