Tarea N°:3

Redes Neuronales Convolucionales

Juan Pablo Miranda Céspedes
Escuela de Ingeniería, Universidad de O'Higgins
11, Noviembre, 2023

***Abstract*—Este proyecto se centra en la implementación de Redes Neuronales Convolucionales (CNN) para la clasificación de imágenes en dos conjuntos de datos ampliamente reconocidos: MNIST y CIFAR-10. Utilizando el lenguaje de programación Python y el marco de trabajo TensorFlow, nuestro objetivo principal es desarrollar un sistema capaz de asignar etiquetas precisas a las imágenes en estos conjuntos de datos.**

Las CNN, también conocidas como ConvNets, son un tipo de red neuronal profunda especialmente diseñadas para el procesamiento de imágenes visuales. Su arquitectura se basa en la idea de pesos compartidos en filtros de convolución que se deslizan a lo largo de las características de entrada, generando mapas de características que son invariantes ante traslaciones.

I. Introducción

El procesamiento de imágenes y la clasificación de objetos en el ámbito de la inteligencia artificial han experimentado avances significativos en los últimos años, gracias al desarrollo de técnicas de aprendizaje profundo. Las Redes Neuronales Convolucionales (CNN) se han convertido en un componente esencial de estas innovaciones y son ampliamente utilizadas en aplicaciones de visión por computadora, como el reconocimiento de imágenes, la segmentación de objetos y el análisis de contenido visual.

En este proyecto, nos enfocamos en la implementación de CNN para abordar un desafío común en el campo de la visión por computadora: la clasificación de imágenes. Nuestro objetivo principal es aplicar estas redes neuronales convolucionales para la clasificación de dos conjuntos de datos icónicos: MNIST y CIFAR-10. Estos conjuntos de datos se han convertido en pilares fundamentales en la evaluación de algoritmos de aprendizaje automático y son reconocidos por su relevancia y complejidad en el campo de la clasificación de imágenes.

La implementación se llevará a cabo utilizando el lenguaje de programación Python y el popular marco de trabajo TensorFlow, que proporciona herramientas poderosas para el desarrollo de redes neuronales. Las CNN son particularmente adecuadas para abordar tareas de clasificación de imágenes, ya que están diseñadas para capturar patrones y características en datos visuales de manera efectiva.

II. Marco Teórico

A. *Convolución.*

La convolución es una forma matemática de combinar dos señales para formar una tercera señal. Es el La técnica más importante en el procesamiento de señales digitales. Usando la estrategia del impulso En descomposición, los sistemas se describen mediante una señal llamada respuesta al impulso. La convolución es importante porque relaciona las tres señales de interés: la señal de entrada, la señal de salida y la respuesta al impulso. Este capítulo presenta la convolución desde dos puntos de vista diferentes, llamados el algoritmo del lado de entrada y el algoritmo del lado de salida..**[1]**

B. *Pooling.*

En el aprendizaje automático y las redes neuronales, las dimensiones de los datos de entrada y los parámetros de la red neuronal desempeñan un papel crucial. Por lo tanto, este número se puede controlar mediante el apilamiento de una o más capas de agrupación.

Dependiendo del tipo de capa de pooling, se realiza una operación en cada canal de los datos de entrada de forma independiente para resumir sus valores en uno solo y así mantener las características más importantes. Estos valores se utilizan como entrada para la siguiente capa del modelo y así sucesivamente. El proceso de agrupación se puede repetir varias veces y cada iteración reduce las dimensiones espaciales. La agregación de valor se puede realizar utilizando diferentes técnicas. **[2]**.

C. *Redes Neuronales Convolucionales*

La red neuronal convolucional (CNN) es un tipo regularizado de red neuronal de retroalimentación que aprende la ingeniería de características por sí misma mediante la optimización de filtros (o kernel). Los gradientes que desaparecen y explotan, observados durante la retropropagación en redes neuronales anteriores, se evitan mediante el uso de pesos regularizados en menos conexiones. Por ejemplo, para cada neurona en la capa completamente conectada se necesitarían 10.000 pesos para procesar una imagen de 100×100 píxeles. Sin embargo, al aplicar núcleos de convolución (o correlación cruzada) en cascada, solo se requieren 25 neuronas para procesar mosaicos de tamaño 5x5. Las características de las capas superiores se extraen de ventanas de contexto más amplias, en comparación con las características de las capas inferiores. **[3]**.

D. *TensorFlow.*

TensorFlow es una plataforma de código abierto de extremo a extremo para el aprendizaje automático. TensorFlow es un sistema

rico para administrar todos los aspectos de un sistema de aprendizaje automático; sin embargo, esta clase se enfoca en el uso de una API de TensorFlow en particular para desarrollar y entrenar modelos de aprendizaje automático. Consulta la documentación de TensorFlow para obtener detalles completos sobre el sistema general de TensorFlow.

Las API de TensorFlow se organizan de manera jerárquica: las API de alto nivel se basan en las de bajo nivel. Los investigadores de aprendizaje automático usan las API de bajo nivel para crear y explorar nuevos algoritmos. En esta clase, usarás una API de alto nivel llamada tf.keras para definir y entrenar modelos de aprendizaje automático y hacer predicciones. tf.keras es la variante de TensorFlow de la API de Keras de código abierto. **[4]**.

E. Adagrad

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates. **[5]**.

F. Pytorch

Pytorch es uno los frameworks para creación de redes neuronales más utilizados de hoy en día. De hecho, tal como se indica en el blog de Assembly (enlace), desde su lanzamiento en Septiembre de 2016, Pytorhc ha pasado de ser utilizado únicamente en el 7 porciento de los papers ha ser utilizado en el 80 porciento de los papers en 2020. **[6]**.

III. Metodología

A. Agregar una nueva dimensión a los conjuntos de prueba y entrenamiento

Luego de cargar la data base Mnist y crear los conjuntos de prueba y entrenamiento, debemos agregar una nueva dimensión a estos conjuntos.

Código 1: Agregar nueva dimensión

```
# Escribe aqu  tu c digo

# Agregar una nueva dimensi n
x_train = np.expand_dims(x_train, axis=3)
x_test = np.expand_dims(x_test, axis=3)

### *No* modifique las siguientes lineas ###
print('Shape of x_train {}'.format(x_train.shape))
print('Shape of y_train {}'.format(y_train.shape))
print('Shape of x_test {}'.format(x_test.shape))
print('Shape of y_test {}'.format(y_test.shape))
```

B. Normalizar imágenes

Dentro de la función, cada valor de píxel en estas imágenes se divide por 255.0. La división por 255.0 escala los valores de píxeles, que típicamente están en el rango de 0 a 255, a un nuevo rango en el que los valores estarán entre 0 y 1. Este tipo de escala es común al trabajar con imágenes, ya que muchos modelos y algoritmos de aprendizaje automático esperan datos en este rango.

Código 2: Normalizar imagenes

```
# Escribe aqu  tu c digo

def normalize_images(images):
    """Normalizar las im genes de entrada.
    """
    # Normalizar la imagen aqu
    images = images / 255.0  # Escalar los valores de p xeles
     en [0, 1]
    return images

### *No* modificar las siguientes l neas ###
test_normalize_images(normalize_images)

# Normalizar los datos para su uso futuro
x_train = normalize_images(x_train)
x_test = normalize_images(x_test)
```

.

1) Configuración de Vector One Hot*:* En este apartado, se nos pide configurar el vector Onehot donde la línea `np.eye(number_classes)` crea una matriz de identidad de tamaño `number_classes`. Esta matriz de identidad es esencialmente una matriz cuadrada con unos en la diagonal principal y ceros en el resto.

La comprensión de lista `[np.eye(number_classes)[int(label)` `for label in vector]` itera sobre cada elemento `label` en el vector de etiquetas. Con `int(label)`, convertimos la etiqueta a un entero, y luego seleccionamos la fila correspondiente de la matriz de identidad. Estas filas se agrupan en una lista llamada `one_hot`.

Finalmente, `return np.array(one_hot)` convierte esa lista de filas en una matriz numpy y la devuelve como el resultado de la función. En resumen, esta parte del código crea una representación one-hot para cada etiqueta en el vector, utilizando una matriz de identidad.

Código 3: Configuración de vector Onehot

```
def one_hot(vector, number_classes):
    """Devuelve una matriz codificada one-hot dado el vector
     argumento.
    """
    # Aqu  almacenaremos nuestros one-hots
    one_hot = []

    # Aqu  se codifica el 'vector' one-hot
    one_hot = one_hot = [np.eye(number_classes)[int(label)] for
      label in vector]
    return np.array(one_hot)


    # Transformar la lista en una matriz numpy y retornarla
    return np.array(one_hot)

### *No* modifique las siguientes lineas ###
test_one_hot(one_hot)

# One-hot codifica los labels de MNIST
y_train = one_hot (y_train, 10)
y_test = one_hot(y_test, 10)
```

C. Definición de conexiones de red y Sample Shape

1) net_1 : Al código base se le agrega las peticiones de la tarea, Definir la entrada de la red para que tenga la dimensión sample shape y crear las conexiones internas de la red

En el código se implementa una red neuronal convolucional (CNN) utilizando la biblioteca Keras. En la primera parte, se define la entrada de la red inputx con dimensiones especificadas por sample shape, que representaría el tamaño y canales de las imágenes de entrada. La segunda parte establece las conexiones internas de la red. Se aplica una capa de convolución con 32 filtros y activación ReLU a la entrada, seguida de una capa de max-pooling para reducir las dimensiones espaciales de la salida convolucional. Posteriormente, la salida se aplana en un vector unidimensional y se conecta a una capa densa completamente conectada con 128 neuronas y activación ReLU. Estas operaciones definen la estructura básica de la CNN, desde la entrada hasta las capas internas.

Código 4: Definición de red y sample shape

```
# Importar la librer a Keras
import keras
from keras.models import Model
from keras.layers import *


def net_1(sample_shape, nb_classes):
    # Defina la entrada de la red para que tenga la dimensi n
     `sample_shape
    input_x = Input(shape=sample_shape)

    # Cree aqu  las conexiones internas de la red
    x = Conv2D(32, kernel_size=(3, 3), activation='relu')(
     input_x)
    x = MaxPooling2D(pool_size=(2, 2))(x)
    x = Flatten()(x)
    x = Dense(128, activation='relu')(x)

    # Dense `nb_classes`
    probabilities = Dense(nb_classes, activation='softmax')(x)

    # Defina la salida
    model = Model(inputs=input_x, outputs=probabilities)

    return model
```

D. Definición de Hiper-parámetros y entrenamiento de la red

En esta parte del código se configura, entrena y evalúa un modelo de red neuronal. Se definen hiperparámetros como **batch size** y **epochs**, y el modelo se compila con la función de pérdida **categorical crossentropy**, el optimizador Adadelta, y se mide la precisión. Posteriormente, el modelo se entrena con los datos de entrenamiento, y se visualizan las curvas de pérdida y precisión. Finalmente, se evalúa el rendimiento en el conjunto de pruebas, mostrando la pérdida y la precisión obtenida en dicho conjunto.

Código 5: Definición de Hiper-Parámetros

```
# Defina los hiperpar metros
batch_size = 64
epochs = 10

### *No* modifique las siguientes l neas ###

# No hay tasa de aprendizaje porque estamos usando los valores
    recomendados
# para el optimizador Adadelta. M s informaci n aqu :
# https://keras.io/optimizers/

# Necesitamos compilar nuestro modelo
model.compile(loss='categorical_crossentropy',
              optimizer='Adadelta',
              metrics=['accuracy'])

# Entrenar
logs = model.fit(x_train, y_train,
                 batch_size=batch_size,
                 epochs=epochs,
                 verbose=2,
                 validation_split=0.1)

# Graficar losses y el accuracy
fig, ax = plt.subplots(1,1)

pd.DataFrame(logs.history).plot(ax=ax)
ax.grid(linestyle='dotted')
ax.legend()

plt.show()

# Evaluar el rendimiento
print('='*80)
print('Assesing Test dataset...')
print('='*80)

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

E. Definición de net 2, sin max pooling

Luego de definir net 1, creamos net 2, a partir de la función anterior. Esta vez eliminaremos la capa de max pooling y agregaremos un STRIDE=2 en el segundo bloque de convolución.

Código 6: Creación net 2

```
#net_2

def net_2(sample_shape, nb_classes):
    # Defina la entrada de la red para que tenga la dimensi n
     `sample_shape`
    input_x = Input(shape=sample_shape)

    # Cree aqu  las conexiones internas de la red sin capa de
     MaxPooling
    x = Conv2D(32, kernel_size=(3, 3), activation='relu')(
     input_x)
    x = Conv2D(32, kernel_size=(3, 3), activation='relu',
     strides=2)(x)
    x = Flatten()(x)
    x = Dense(128, activation='relu')(x)

    # Dense `nb_classes`
    probabilities = Dense(nb_classes, activation='softmax')(x)

    # Defina la salida
    model = Model(inputs=input_x, outputs=probabilities)

    return model
```

F. Creación de conjuntos de prueba y entrenamiento usando one hot

Para este apartado, debemos usar la función one hot anteriormente definida, para crear y_{train} y y_{test}

Código 7: Creación de conjuntos usando One Hot

```
num = 10
y_train = one_hot(y_train, 10)
y_test = one_hot(y_test, 10)

```

```
### *No* modifique las siguientes l neas ###
# Imprima los tama os de los datos (variables)
print('Shape of x_train {}'.format(x_train.shape))
print('Shape of y_train {}'.format(y_train.shape))
print('Shape of x_test {}'.format(x_train.shape))
print('Shape of y_test {}'.format(y_train.shape))
```

G. Normalización de imágenes

Luego, usando la función NORMALIZE_IMAGES, normalizamos los conjuntos de prueba y entrenamiento.

Código 8: Normalización de imágenes

```
x_test = normalize_images(x_test)
x_train = normalize_images(x_train)
```

H. Creación y entrenamiento de modelo

Se crea un código para entrenar una red neuronal con 30 épocas, BATCH_SIZE de 128, VALIDATION_SPLIT de 0.2

Código 9: Red Neuronal con respectiva configuración

```
import keras
from keras.datasets import cifar10
from sklearn.model_selection import train_test_split
from keras.callbacks import EarlyStopping, ModelCheckpoint

# Dividir los datos en conjuntos de entrenamiento y validaci n
x_train, x_val, y_train, y_val = train_test_split(x_train,
    y_train, test_size=0.2, random_state=42)

# Definir los callbacks para el entrenamiento
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
     verbose=1, restore_best_weights=True)
model_checkpoint = ModelCheckpoint('best_model.h5', monitor='
    val_loss', save_best_only=True, verbose=1)

# Entrenar el modelo con 30  pocas , batch size de 128 y
    validation_split de 0.2
history = model.fit(x_train, y_train, epochs=30, batch_size
    =128, validation_split=0.2, callbacks=[early_stopping,
    model_checkpoint])

# Evaluar el rendimiento en el conjunto de pruebas
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'P rdida en el conjunto de pruebas: {test_loss}')
print(f'Precisi n en el conjunto de pruebas: {test_accuracy}')
```

IV. Resultados

A. Defina los hiper-parámetros y entrena la red

Dado el parámetro ÉPOCAS y BATCH_SIZE , configuramos la red con 20 épocas y 128 BATCH_SIZE. Además de esto, considerar que se usa net1 .

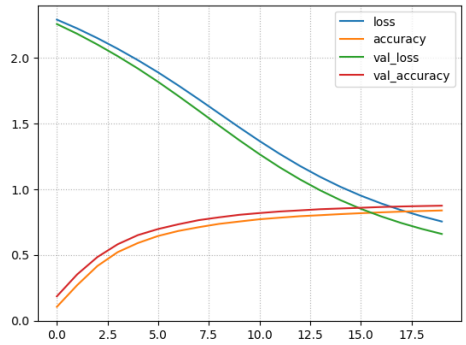

Fig. 1: 20 épocas y 128 de Batch size.

- Test loss: 0.70
- Test Accuracy: 86%

Los valores para la ultima época son:

Época	Pasos	Tiempo/Epoch	Pérdida	Precisión
20	422	2s	0.7535	0.8374

TABLE I: Resumen de entrenamiento.

Época	Pasos	Tiempo/Epoch	Pérdida Val	Precisión Val
20	-	2s/epoch - 4ms/step	0.6588	0.8738

TABLE II: Resumen de validación.

B. Defina los hiper-parámetros y entrena la red sin max pooling

Dado el parámetro ÉPOCAS y BATCH_SIZE , configuramos la red con 20 épocas y 128 BATCH_SIZE.

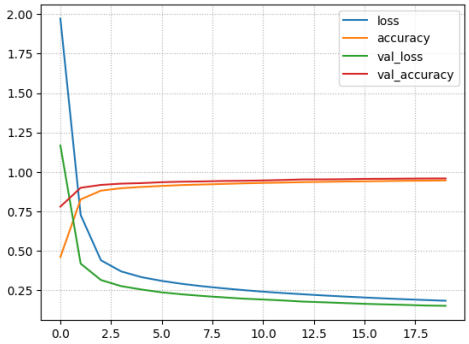

Fig. 2: 20 épocas y 128 de Batch size, sin max pooling.

- Test loss: 0.17%
- Test Accuracy: 95%

Los valores para la última época son:

Época	Pasos	Tiempo/Epoch	Pérdida	Precisión
20	422	2s	0.1835	0.9460

TABLE III: Resumen de entrenamiento.

Época	Pasos	Tiempo/Epoch	Pérdida Val	Precisión Val
20	-	2s/epoch - 5ms/step	0.1514	0.9587

TABLE IV: Resumen de validación.

C. *Comparación entre net_1 y net_2*

Primeramente, compararemos net_1 y net_2 usando Adagrad como optimizador, para elegir cual de los dos modelos funciona mejor:

1) NET_1 CON ADAGRAD*:* Para esta configuración, mostramos los resultados.

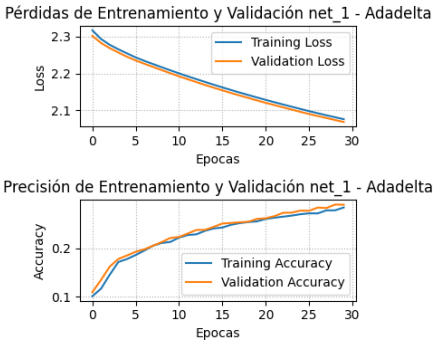

Fig. 3: net_1 con Adagrad.

- Test loss: 1.67
- Test Accuracy: 43%

Los valores para la última épocas son:

Época	Pasos	Tiempo/Epoch	Pérdida	Precisión
30	200	1s	1.6607	0.4343

TABLE V: Resumen de entrenamiento.

Época	Pasos	Tiempo/Epoch	Pérdida Val	Precisión Val
30	313	1s/epoch - 3ms/step	1.6819	0.4192

TABLE VI: Resumen de validación.

2) NET_2 CON ADAGRAD*:* Para esta configuración, mostramos los resultados.

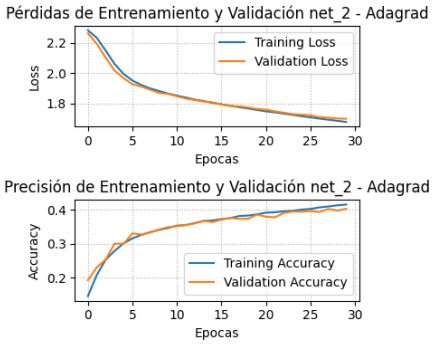

Fig. 4: net_2 con Adagrad.

- Test loss: 1.70
- Test Accuracy: 41%

Los valores para la última épocas son:

Época	Pasos	Tiempo/Epoch	Pérdida	Precisión
30	160	1s	1.6774	0.4157

TABLE VII: Resumen de entrenamiento.

Época	Pasos	Tiempo/Epoch	Pérdida Val	Precisión Val
30	313	1s/epoch - 4ms/step	1.6987	0.4031

TABLE VIII: Resumen de validación.

D. ENTRENAR MODELO CIFAR-10 CON NET_1

Para este apartado se nos pide entrenar una CNN para CIFAR-10. Esto se debe hacer utilizando net_1, dados los resultados obtenidos.

1) Entrenamiento 1: Entrenamos el modelo con los siguientes parámetros:

- Función: net_1
- Épocas: 30.
- Batch_size : 128.
- Validation_split: 0.2.
- Optimizador: Adagrad.

Los resultados obtenidos son los siguientes:

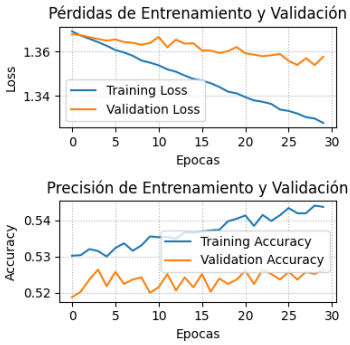

Fig. 5: CIFAR-10 Y Adagrad

- Pérdida en el conjunto de pruebas: 1.40.
- Precisión en el conjunto de pruebas: 51%.

Los resultados para la ultima época son:

Epoch	Condiciones	Pérdida	Precisión
30	No Mejora	1.3277	0.5438

TABLE IX: Resultados del Modelo en Época 30.

Conjunto	Tiempo/Epoch	Pérdida	Precisión
Entrenamiento	1s	1.3277	0.5438
Validación	1s	1.3576	0.5264
Test	1s	1.3967	0.5147

TABLE X: Detalles del Entrenamiento y Evaluación.

2) Entrenamiento 2: Entrenamos el modelo con los siguientes parámetros:

- Función: net_1
- Épocas: 30.
- Batch_size : 128.
- Validation_split: 0.2.
- Optimizador: Adadelta.

Los resultados obtenidos son los siguientes:

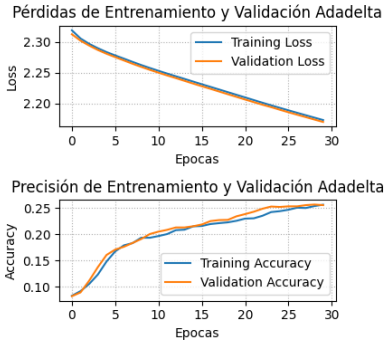

Fig. 6: CIFAR-10 Y Adadelta

- Pérdida en el conjunto de pruebas: 2.17.
- Precisión en el conjunto de pruebas: 25%.

Los resultados para la ultima época son:

Epoch	Condiciones	Pérdida	Precisión
30	Mejora	2.1727	0.2567

TABLE XI: Resultados del Modelo en Época 30.

Conjunto	Pérdida	Precisión
Entrenamiento	2.1727	0.2567
Validación	2.1696	0.2559
Test	2.17198	0.2526

TABLE XII: Detalles del Entrenamiento y Evaluación.

3) Entrenamiento 3: Entrenamos el modelo con los siguientes parámetros:

- Función: net_1
- Épocas: 30.
- Batch_size : 128.
- Validation_split: 0.2.
- Optimizador: Adam.

Los resultados obtenidos son los siguientes:

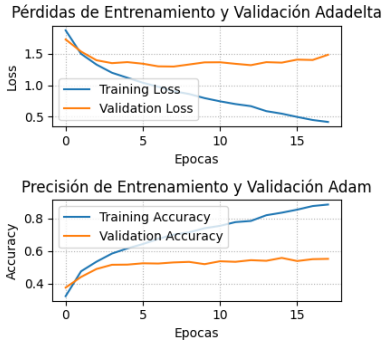

Fig. 7: CIFAR-10 Y Adam

- Pérdida en el conjunto de pruebas: 1.27.
- Precisión en el conjunto de pruebas: 56%.

Los resultados para la ultima época son(Hubo Early Stopping):

Epoch	Condiciones	Pérdida	Precisión
18	Detención temprana	1.2724	0.5585

TABLE XIII: Resultados del Modelo en Época 18.

Conjunto	Valor
Test	1.27235
Precisión en el conjunto de pruebas	0.5585

TABLE XIV: Detalles del Entrenamiento y Evaluación.

V. Análisis

En virtud de los resultados mostrados en el anterior apartado, se deben hacer los análisis correspondientes para así, interpretar los resultados en base al mejor modelo y/o configuración que se iteró en la investigación.

A. ANÁLISIS DE LAS CNN CON Y SIN MAX POOLING

Se evaluaron dos modelos de redes neuronales convolucionales (CNN) en el conjunto de datos MNIST, net_1 con max pooling y net_2 sin max pooling.

Modelo net_1 con Max Pooling

El modelo net_1 con max pooling presenta resultados satisfactorios en el conjunto de prueba. La pérdida en la última época es de **0.7535**, y la precisión alcanza un **86%**. Estos resultados indican que el modelo tiene un rendimiento aceptable, aunque puede haber margen para mejorar la precisión.

Modelo net_2 sin Max Pooling

El modelo net_2 sin max pooling demuestra un rendimiento notablemente mejor en comparación con net_1. La pérdida en la última época es significativamente menor, alcanzando **0.1835**, y la precisión es más alta, llegando al **95%** en el conjunto de prueba. Además, en validación, la precisión es **95.87%**, mostrando una capacidad sólida de generalización.

B. ¿QUÉ MODELO FUNCIONA MEJOR?

Dada la comparación de modelos, mostramos un resumen de las métricas obtenidas por cada uno;

Modelo	Precisión	Pérdida
net_1	43%	1.67
net_2	41%	1.70

TABLE XV: Comparación de modelos.

Después de examinar detenidamente los resultados obtenidos para **net_1** y **net_2** tras 30 épocas de entrenamiento, se observa que ambos modelos presentan un rendimiento comparable en términos de métricas clave. **net_1** exhibe una precisión de entrenamiento del **43.43%**, una precisión de validación del **41.92%**, y pérdidas de entrenamiento y validación de **1.6607** y **1.6819**, respectivamente. Por otro lado, **net_2** presenta una precisión de entrenamiento del **41.57%**, una precisión de validación del **40.31%**, y pérdidas de entrenamiento y validación de **1.6774** y **1.6987**, respectivamente.

Aunque **net_1** evidencia un desempeño ligeramente superior en todas las métricas evaluadas, es crucial considerar que la disparidad entre ambos modelos no es substancial. La elección del modelo óptimo puede depender de los requisitos específicos de la aplicación y la importancia atribuida a métricas particulares, como la precisión o la pérdida en el conjunto de validación. Es posible que, en escenarios prácticos, se requiera un ajuste adicional de hiperparámetros o la exploración de arquitecturas alternativas para discernir mejoras significativas en el rendimiento. En conclusión, en función de los resultados presentados, **net_1** se perfila como la opción preferida debido a su desempeño ligeramente superior.

C. ¿QUÉ OPTIMIZADOR FUNCIONA MEJOR?

En el análisis comparativo de los tres optimizadores, Adagrad, Adadelta y Adam, se observa que Adam ha demostrado ser el más eficaz en términos de rendimiento. En el conjunto de entrenamiento, Adam logra una pérdida de 1.2724 con una precisión del 55.85%, superando a Adagrad, que alcanza una pérdida de 1.3277 y una precisión del 54.38%. Adadelta, por otro lado, muestra un rendimiento inferior con una pérdida de 2.1727 y una precisión del 25.67%. Este patrón se mantiene consistente en los conjuntos de validación y prueba, donde Adam demuestra una menor pérdida y una mayor precisión en comparación con los otros optimizadores. La consistencia en el rendimiento de Adam, junto con su menor pérdida y mayor precisión, sugiere que es la opción más adecuada entre los tres optimizadores evaluados para este problema específico.

D. ¿EXISTE ALGUNA EVIDENCIA DE OVERFITTING?

Al analizar los resultados obtenidos, no parece haber evidencia clara de overfitting en el modelo. La pérdida en el conjunto de entrenamiento es comparable a la pérdida en el conjunto de prueba para los optimizadores Adagrad y Adam. Sin embargo, es importante señalar que la precisión en el conjunto de entrenamiento es ligeramente superior en comparación con el conjunto de prueba para Adagrad y Adam. Este fenómeno puede indicar una ligera sobreoptimización para los datos de entrenamiento. En el caso de Adadelta, donde la precisión es considerablemente más baja, podría sugerir subajuste o que el modelo no es lo suficientemente complejo para capturar la variabilidad en los datos de entrenamiento. En resumen, aunque hay indicios de una ligera sobreoptimización en algunos casos, la presencia de overfitting no parece ser significativa en base a la diferencia entre el rendimiento en los conjuntos de entrenamiento y prueba.

E. ¿CÓMO PODEMOS MEJORAR AÚN MÁS EL RENDIMIENTO?.

1) Sobre los modelos net_1 y net_2 : Según los resultados de las redes neuronales net_1 y net_2 después de 30 épocas, se observa que ambas presentan un rendimiento subóptimo en términos de precisión y pérdida en los conjuntos de entrenamiento y validación. La precisión de entrenamiento para net_1 es del **43.43%**, mientras que para net_2 es del **41.57%**. Además, las pérdidas de entrenamiento para ambas redes son relativamente altas, con **1.6607** para net_1 y **1.6774** para net_2.

Para mejorar el rendimiento, se podrían explorar varias estrategias. En primer lugar, sería recomendable ajustar los hiperparámetros del modelo, como la tasa de aprendizaje, la arquitectura de la red y el tamaño del lote, mediante técnicas de búsqueda sistemática. Además, se podría considerar la posibilidad de aumentar la complejidad de la red neuronal, por ejemplo, añadiendo capas o unidades, para capturar patrones más complejos en los datos. La regularización, mediante técnicas como la dropout, también podría ayudar a prevenir el sobreajuste y mejorar la generalización.

Es importante destacar que este proceso de mejora debe ir acompañado de un monitoreo continuo del rendimiento en conjuntos de validación y prueba para evitar el sobreajuste. La combinación de ajuste de hiperparámetros, cambios en la arquitectura y técnicas de regularización puede ser crucial para lograr un rendimiento óptimo en estas redes neuronales.

2) Sobre los 3 optimizadores: Con respecto a los resultados de los tres optimizadores, Adagrad, Adadelta y Adam, y la posibilidad de mejorar su rendimiento, se identifican áreas para mejorar el rendimiento. En el caso de Adagrad, se observa una precisión en el conjunto de entrenamiento del **54.38%**, pero una precisión de validación y prueba ligeramente más baja (**52.64%** y **51.47%** respectivamente). Para Adadelta, la precisión en el conjunto de entrenamiento es significativamente más baja (**25.67%**), lo que indica un rendimiento subóptimo. Adam, aunque muestra el mejor rendimiento entre los tres optimizadores, todavía podría mejorarse, ya que la precisión en el conjunto de entrenamiento y prueba es del **55.85%**.

Para mejorar el rendimiento, se podría considerar ajustar los hiperparámetros de cada optimizador, como la tasa de aprendizaje y los momentos específicos. Además, la arquitectura de la red y las técnicas de regularización podrían ser exploradas para aumentar la capacidad de generalización del modelo. La búsqueda sistemática de hiperparámetros y la experimentación con arquitecturas de red más complejas son pasos clave para optimizar el rendimiento de estas configuraciones. Se destaca la importancia de monitorear el rendimiento en conjuntos de validación y prueba para evitar el sobreajuste.

VI. Conclusiones Generales

Comparando ambos modelos(net_1 y net_2, con y sin max pooling respectivamente), net_2 sin max pooling supera significativamente a net_1 en términos de precisión y pérdida en el conjunto de prueba. La ausencia de max pooling en net_2 parece haber contribuido a una mejor capacidad de aprendizaje de características intrincadas en los datos. Es importante destacar que la precisión del **95%** en el conjunto de prueba de net_2 indica un rendimiento muy bueno en la tarea de clasificación de dígitos MNIST.

Considerando los resultados de los modelos de redes neuronales (net_1 y net_2) y los tres optimizadores (Adagrad, Adadelta y Adam), se pueden extraer algunas conclusiones cruciales.

En el caso de los modelos net_1 y net_2, ambos presentan un rendimiento subóptimo con precisiones de entrenamiento del **43.43%** y **41.57%** respectivamente. Además, las pérdidas de entrenamiento son relativamente altas, con valores de **1.6607** y **1.6774** respectivamente. Para mejorar estos modelos, se sugiere la ajustar hiperparámetros, considerar cambios en la arquitectura de la red y aplicar técnicas de regularización.

En cuanto a los optimizadores, Adam destaca como el más efectivo entre los tres, con una precisión de entrenamiento del **55.85%**. Adagrad muestra un rendimiento aceptable, mientras que Adadelta tiene una precisión de entrenamiento notablemente más baja (**25.67%**). Para mejorar el rendimiento general, una opción es ajustar los hiperparámetros de cada optimizador y realizar experimentos sistemáticos para encontrar la configuración más efectiva.

En resumen, el proceso de mejora debe abordar tanto los modelos individuales como la elección del optimizador. La optimización de hiperparámetros y la experimentación con arquitecturas de red más complejas son pasos esenciales para lograr un rendimiento óptimo. Además, se destaca la importancia de monitorear el rendimiento en conjuntos de validación y prueba para evitar problemas de sobreajuste.

VII. Referencias

1) **[1]** https://www.analog.com/media/en/technical-documentation/dsp-book/dsp_book_ch6.pdf
2) **[2]** https://www.baeldung.com/cs/neural-networks-pooling-layers#:~:text=Introduction,relationships%20in%20the%20input%20data.
3) **[3]** https://en.wikipedia.org/wiki/Convolutional_neural_network
4) **[4]** https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit?hl=es-419#:~:text=TensorFlow%20is%20an%20end-to,and%20train%20machine%20learning%20models.
5) **[5]** https://keras.io/api/optimizers/adagrad/
6) **[6]** https://www.edx.org/es/aprende/pytorch#:~:text=Qu%20es%20PyTorch%3F,en%20Python%2C%20C%2B%2B%20y%20CUDA.