Tarea N°:2

Redes Neuronales

Juan Pablo Miranda Céspedes
Escuela de Ingeniería, Universidad de O'Higgins
21, Octubre, 2023

***Abstract*—Este estudio se centra en el entrenamiento de redes neuronales para la clasificación de dígitos utilizando el conjunto de datos Optical Recognition of Handwritten Digits. Con 64 características, 10 clases y 5620 muestras, este conjunto de datos desafía la capacidad de las redes neuronales para aprender patrones de escritura a mano. La estructura de las redes a entrenar consta de una capa de entrada de 64 dimensiones, capas ocultas (una o dos) y una capa de salida con 10 neuronas, empleando la función de activación softmax.**

El uso de la función de pérdida de entropía cruzada y el optimizador Adam garantiza un aprendizaje eficiente. Es importante destacar que la función softmax está implícita en la función de pérdida CrossEntropyLoss de PyTorch, evitando la necesidad de agregarla manualmente en la salida de la red.

Mediante este análisis exhaustivo, se pretende arrojar luz sobre las mejores prácticas en la configuración de redes neuronales para la clasificación de dígitos escritos a mano, lo que podría tener un impacto significativo en la precisión y eficiencia de los sistemas de reconocimiento de escritura automatizados.

I. Introducción

La presente tarea tiene como objetivo explorar y analizar el desempeño de redes neuronales en la clasificación de dígitos a través del uso del conjunto de datos **"'Optical Recognition of Handwritten Digits."** Para alcanzar este propósito, se realizarán una serie de experimentos utilizando distintas configuraciones de redes neuronales y se evaluará su rendimiento en conjuntos de entrenamiento, validación y prueba.

Para cada red entrenada, se realizarán análisis exhaustivos que incluyen la medición del tiempo de entrenamiento, la representación gráfica de las curvas de pérdida de entrenamiento y validación, así como la generación de matrices de confusión normalizadas y la evaluación de la precisión utilizando los conjuntos de entrenamiento y validación.

Finalmente, se identificará la mejor red en función de su precisión en el conjunto de validación y se evaluará su rendimiento en el conjunto de prueba. Se llevará a cabo un análisis detallado de los resultados obtenidos, explorando cómo la cantidad de neuronas en la capa oculta, la cantidad de capas ocultas y la función de activación influyen en el rendimiento de la red. Se compararán los tiempos de entrenamiento y se analizarán las matrices de confusión y las precisiones en diferentes configuraciones de la red en el conjunto de validación. Además, se compararán los resultados en el conjunto de prueba con los obtenidos en el conjunto de validación.

Este estudio proporcionará una comprensión más profunda de cómo la arquitectura de la red neuronal y sus parámetros afectan el rendimiento en tareas de clasificación, así como las mejores prácticas para evitar el sobreajuste.

II. Marco Teórico

A. *Redes Neuronales.*

En resumidas palabras, Una red neuronal es un método de la inteligencia artificial que enseña a las computadoras a procesar datos de una manera que está inspirada en la forma en que lo hace el cerebro humano**[1]**. El cerebro humano inspira la arquitectura de las redes neuronales. Las neuronas, forman una red compleja y con un alto nivel de interconexión y se envían señales eléctricas entre sí para ayudar a los humanos a procesar la información. De manera similar, una red neuronal artificial está formada por neuronas artificiales que trabajan juntas para resolver un problema.**[2]**

B. *Capa oculta.*

En las redes neuronales, la capa oculta se encuentra entre la entrada y la salida del algoritmo, en la cual la función aplica pesos a las entradas y las dirige a través de una función de activación como salida. En resumen, las capas ocultas realizan transformaciones no lineales de las entradas introducidas en la red. Las capas ocultas varían según la función de la red neuronal y, de manera similar, las capas pueden variar según los pesos asociados a ellas. **[3]**.

C. *Función de perdida*

Las funciones de pérdida ocupan un lugar de trascendental importancia en el contexto de las redes neuronales, dado que se erigen como un elemento de responsabilidad ineludible junto a las funciones de optimización, encargadas de conformar el ajuste del modelo a los datos de entrenamiento provistos. **[4]**.

D. *Métricas de evaluación.*

Explora las métricas utilizadas para evaluar el rendimiento de un modelo de red neuronal. Incluye conceptos como precisión, recall, F1-score, y la curva ROC-AUC. Explica cómo estas métricas permiten medir la eficacia de un modelo en la clasificación de datos.

E. Matriz de confusión

Una matriz de confusión es un resumen tabular de la cantidad de predicciones correctas e incorrectas realizadas por un clasificador. Se utiliza para medir el rendimiento de un modelo de clasificación. Puede utilizarse para evaluar el rendimiento de un modelo de clasificación mediante el cálculo de métricas de rendimiento como precisión, exactitud, recuperación (recall) y puntuación F1. **[5]**.

F. Normalización de datos

La normalización es una técnica de preparación de datos que se utiliza con regularidad en el aprendizaje automático. El procedimiento de ajustar las columnas de un conjunto de datos al mismo rango se conoce como normalización. No todos los conjuntos de datos requieren ser normalizados para el aprendizaje automático; esto solo es necesario cuando las escalas de las características presentan diferencias significativas. **[6]**.

III. Metodología

A. Preprocesamiento de datos

Primeramente, se cargó la base de datos desde un repositorio de GitHub.

Código 1: carga de DB

```
!wget https://raw.githubusercontent.com/Felipe1401/Mineria/main/dataset_digits/l_digits_train.txt
!wget https://raw.githubusercontent.com/Felipe1401/Mineria/main/dataset_digits/l_digits_test.txt
```

Gracias a esto, podemos cargar los datos sin necesidad de extraerlos desde el directorio del computador en que estemos corriendo el código.

Posteriormente, se cargan los conjuntos de entrenamiento y prueba.

Código 2: carga de conjuntos

```
# Cargar conjuntos de datos de entrenamiento y prueba
column_names = ["feat" + str(i) for i in range(64)]
column_names.append("class")

df_train_val = pd.read_csv('l_digits_train.txt', names = column_names)
df_test = pd.read_csv('l_digits_test.txt', names = column_names)
```

B. Creación de conjuntos de entrenamiento y validación.

En este apartado, se define la creación de los conjuntos de entrenamiento y validación que se usarán desde este punto en adelante para el desarrollo de la tarea.

Código 3: Código para definir conjuntos de entrenamiento y validación

```
df_train, df_val = train_test_split(df_train_val, test_size = 0.3, random_state = 10)
print("Muestras de entrenamiento: ", len(df_train))
print("Muestras de validaci n: ", len(df_val))
print("Muestras de prueba: ", len(df_test))
print("Muestras totales: ", len(df_train_val)+len(df_test))
```

Posteriror a esto, se normalizan los datos para poder trabajar con ellos.

1) Creación de Dataloaders para Pytorch*:* Para poder usar Pytorch y su ecosistema, adecuaremos los datos como dataloaders.

Código 4: Crear dataloaders para Pytorch

```
# Crear datasets
feats_train = df_train.to_numpy()[:,0:64].astype(np.float32)
labels_train = df_train.to_numpy()[:,64].astype(int)
dataset_train = [ {"features":feats_train[i,:], "labels": labels_train[i]} for i in range(feats_train.shape[0]) ]

feats_val = df_val.to_numpy()[:,0:64].astype(np.float32)
labels_val = df_val.to_numpy()[:,64].astype(int)
dataset_val = [ {"features":feats_val[i,:], "labels":labels_val[i]} for i in range(feats_val.shape[0]) ]

feats_test = df_test.to_numpy()[:,0:64].astype(np.float32)
labels_test = df_test.to_numpy()[:,64].astype(int)
dataset_test = [ {"features":feats_test[i,:], "labels": labels_test[i]} for i in range(feats_test.shape[0]) ]


#Creamos dataloaders

# Crear dataloaders
dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=128, shuffle=True, num_workers=0)
dataloader_val = torch.utils.data.DataLoader(dataset_val, batch_size=128, shuffle=True, num_workers=0)
dataloader_test = torch.utils.data.DataLoader(dataset_test, batch_size=128, shuffle=True, num_workers=0)
```

C. Definición de Red Neuronal

Luego, en esta parte del código se define un **modelo de red neuronal** con una **capa oculta** compuesta por **10 neuronas** y utiliza la función de activación **ReLU**. El modelo se construye mediante la clase `nn.Sequential` de PyTorch e incluye una **capa de entrada** con 64 unidades, seguida de la activación ReLU y una **capa de salida** con 10 unidades.

Luego, se determina el **dispositivo de ejecución** en el que se correrá el modelo. Si una **GPU** está disponible, se selecciona "cuda"; de lo contrario, se utiliza la **CPU**.

Se definen la **función de pérdida** (`criterion`) y el **optimizador** (`optimizer`) para el entrenamiento. En este caso, la función de pérdida es **CrossEntropyLoss** y el optimizador es **Adam**, con una tasa de aprendizaje de **0.001**.

El entrenamiento de la red se realiza a lo largo de un máximo de **1000 épocas**. Se registran las **pérdidas de entrenamiento** y **validación**, así como el **accuracy de validación** en lotes, en listas correspondientes. Además, se implementa la técnica de "**early stopping**" para evitar el sobreajuste, donde se controla el número máximo de **épocas sin mejora** (definido por `patience`) y se detiene el entrenamiento si se supera este límite.

Este bucle de entrenamiento abarca todas las épocas y realiza el seguimiento de las **métricas de rendimiento** en cada lote.

Código 5: Definición de red

```
# -- Modelo con una capa oculta de 10 neuronas y activaci n ReLU --
model = nn.Sequential(
    nn.Linear(64, 10),  # Capa de entrada
    nn.ReLU(),          # Funci n de activaci n ReLU
    nn.Linear(10, 10)   # Capa de salida
)
```

```

# Se le indica a PyTorch que correremos el modelo en GPU si est disponible
device = "cuda" if torch.cuda.is_available() else "cpu"
model = model.to(device)

# Definimos la funci n de p rdida y el optimizador que utilizaremos
criterion = nn.CrossEntropyLoss()                                # Funci n de p rdida
optimizer = optim.Adam(model.parameters(), lr=1e-3)     # Optimizador

start = time.time()

# Guardar resultados del loss y las  pocas  que dur  el entrenamiento
loss_train = []
loss_val = []
accuracy_val = []  # Lista para guardar el accuracy de validaci n
epochs = []

# Entrenamiento de la red por un m ximo de 1000  pocas
max_epochs = 1000

# Variables para el early stopping
best_val_loss = float('inf')
patience = 10  # N mero m ximo de  pocas  sin mejora permitidas
counter = 0

for epoch in range(max_epochs):
    # Guardar loss y accuracy de cada batch
    loss_train_batches = []
    loss_val_batches = []
    accuracy_val_batches = []  # Lista para guardar el accuracy de validaci n por lotes
```

D. Entrenamiento, validación y resultados

El siguiente fragmento de código representa la fase de **entrenamiento** y **validación** de una red neuronal:

Primero, se coloca el modelo en el modo de **entrenamiento** con `model.train()`. Luego, se procede a recorrer cada **batch** de los datos de entrenamiento, y para cada batch, se realizan las siguientes tareas:

- Las características y las etiquetas del batch actual se transfieren al dispositivo de ejecución especificado por `device`. - Se establece el gradiente del optimizador en cero con `optimizer.zero_grad()`. - El modelo realiza una **predicción** con las características de entrada y se calcula el **loss** utilizando la función de pérdida definida. - El gradiente se propaga hacia atrás con `loss.backward()`. - El optimizador ajusta los **pesos de la red** con `optimizer.step()`. - La pérdida del batch actual se registra en la lista `loss_train_batches`.

Al final del bucle de entrenamiento, se calcula y almacena el **loss promedio de entrenamiento** para la época actual en la lista `loss_train`.

Luego, se inicia la fase de **validación** con `model.eval()`. Se recorren los datos de validación en lotes y, para cada lote, se realizan tareas similares a las de entrenamiento, incluyendo la evaluación del loss de validación y el cálculo del **accuracy** de validación. Estos valores se almacenan en las listas `loss_val_batches` y `accuracy_val_batches`.

Posteriormente, se calcula el **loss promedio de validación** y el **accuracy promedio de validación** para la época actual y se almacenan en las listas `loss_val` y `accuracy_val`, respectivamente.

Se registra el número de **época** actual en la lista `epochs`.

Se imprime en la consola el **loss de entrenamiento**, el **loss de validación** y el **accuracy de validación** para la época actual.

Para implementar el **early stopping**, se compara el loss de validación actual con el mejor loss de validación previamente registrado. Si no hay mejora, un contador llamado `counter` se incrementa en 1. Si el contador supera un umbral definido por `patience`, el entrenamiento se detiene y se muestra un mensaje de detención.

Finalmente, se registra el tiempo total que tomó el entrenamiento y se muestra en la consola.

Código 6: Entrenamiento y validación

```
    # Entrenamiento
    ------------------------------------------------------------------

    model.train()
    # Debemos recorrer cada batch (lote de los datos)
    for i, data in enumerate(dataloader_train, 0):
        # Procesar el batch actual
        inputs = data["features"].to(device)  # Caracter sticas
        labels = data["labels"].to(device)    # Clases
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        loss_train_batches.append(loss.item())

    # Guardamos el loss de entrenamiento de la  poca  actual
    loss_train.append(np.mean(loss_train_batches))

    # Validaci n
    ------------------------------------------------------------------

    model.eval()
    with torch.no_grad():
        for i, data in enumerate(dataloader_val, 0):
            inputs = data["features"].to(device)
            labels = data["labels"].to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss_val_batches.append(loss.item())

            # Calcula las predicciones del modelo
            _, predicted = torch.max(outputs, 1)

            # Convierte las etiquetas y predicciones al formato de CPU para calcular el accuracy
            labels_cpu = labels.cpu().numpy()
            predicted_cpu = predicted.cpu().numpy()

            accuracy = accuracy_score(labels_cpu, predicted_cpu)
            accuracy_val_batches.append(accuracy)

    # Guardamos el Loss de validaci n y el accuracy de validaci n de la  poca  actual
    loss_val.append(np.mean(loss_val_batches))
    accuracy_val.append(np.mean(accuracy_val_batches))

    # Guardamos la  poca
    epochs.append(epoch)

    # Imprimir la p rdida de entrenamiento/validaci n y el accuracy de validaci n en la  poca  actual
    print(f"Epoch: {epoch}, train loss: {loss_train[epoch]:.4f}, val loss: {loss_val[epoch]:.4f}, val accuracy: {accuracy_val[epoch]:.4f}")
```

```
    # Early Stopping: Verificar si la p rdida de validaci n ha dejado de mejorar
    if loss_val[epoch] < best_val_loss:
        best_val_loss = loss_val[epoch]
        counter = 0  # Reiniciar el contador
    else:
        counter += 1

    if counter >= patience:
        print("Deteniendo el entrenamiento debido a falta de mejora en la validaci n.")
        break  # Salir del bucle de entrenamiento

end = time.time()
print(f'Finished Training, total time {end - start:.2f} seconds')
```

E. Iteración para cada escenario

Dado el código mencionado y explicado anteriormente, para cada escenario se realizaran configuraciones diferentes. En cada escenario, el numero de neuronas en la capa oculta, el numero de capas ocultas o la función de activación varían. Usamos el mismo código, con la salvedad de que para cada caso, se definen diferentes configuraciones.

Esta diferencia, se muestra en los siguientes códigos, uno para cada escenario.

Código 7: config escenario A

```
# -- Modelo con una capa oculta de 10 neuronas y activaci n ReLU --
model = nn.Sequential(
    nn.Linear(64, 10),  # Capa de entrada
    nn.ReLU(),          # Funci n de activaci n ReLU
    nn.Linear(10, 10)   # Capa de salida
)
```

Código 8: config escenario B

```
# -- Modelo con una capa oculta de 40 neuronas y activaci n ReLU --
model = nn.Sequential(
    nn.Linear(64, 40),  # Capa de entrada
    nn.ReLU(),          # Funci n de activaci n ReLU
    nn.Linear(40, 10)   # Capa de salida
)
```

Estas dos configuraciones de red neuronal difieren en la cantidad de neuronas en la capa oculta y en la estructura de la red. En el primer modelo, se utiliza una capa oculta con 10 neuronas, mientras que en el segundo modelo, se emplea una capa oculta con 40 neuronas. Ambos modelos comparten una capa de entrada con 64 unidades y utilizan la función de activación ReLU en la capa oculta, seguida por una capa de salida con 10 unidades en ambas configuraciones.

Para cada escenario a iterar, será necesario cambiar los parámetros en esta sección del código

F. Matrices de confusión

Para esta parte de la tarea se realizan una serie de tareas relacionadas con la evaluación del rendimiento del modelo de aprendizaje automático, particularmente en un escenario de clasificación. A continuación, se describen las principales acciones:

Primero, se define una función denominada **get_predictions_and_labels** que toma como entrada un **modelo**, un **dataloader** (un objeto que facilita el acceso a los datos) y un **dispositivo** de ejecución (por ejemplo, CPU o GPU). La función tiene como objetivo obtener las **predicciones** del modelo y las **etiquetas reales** para un conjunto de datos dado. El modelo se coloca en el modo de **evaluación** utilizando `model.eval()` para garantizar que no se realicen ajustes durante esta fase. Luego, se recorren los lotes de datos, se realizan las predicciones y se almacenan tanto las predicciones como las etiquetas reales.

A continuación, se utilizan estas predicciones y etiquetas para dos conjuntos de datos diferentes: el **conjunto de validación** y el **conjunto de entrenamiento**. Se llaman a la función `get_predictions_and_labels` para ambos conjuntos, lo que resulta en las predicciones y etiquetas reales para cada uno.

Luego, se calculan las **matrices de confusión** para el conjunto de validación y el conjunto de entrenamiento utilizando las predicciones y etiquetas reales. Las matrices de confusión proporcionan información sobre cómo se clasifican las muestras en diferentes categorías.

A continuación, se crea una figura con dos subgráficos para visualizar las matrices de confusión. La primera figura muestra la matriz de confusión del **conjunto de validación**, mientras que la segunda figura muestra la matriz de confusión del **conjunto de entrenamiento**. Estas visualizaciones ayudan a comprender cómo se distribuyen las predicciones en comparación con las etiquetas reales.

Finalmente, se calculan y muestran las **precisiones** (accuracy) tanto para el **conjunto de entrenamiento** como para el **conjunto de validación**. La precisión es una medida de cuántas predicciones acertadas se hicieron en relación con el total de muestras.

Para cada escenario, correremos este código para evaluar la precesión del modelo.

Código 9: Matrices de confusión

```
# Obtener predicciones y etiquetas para un conjunto dado
def get_predictions_and_labels(model, dataloader, device):
    preds = []
    labels = []

    model.eval()  # Cambiar el modo del modelo a evaluaci n
    with torch.no_grad():
        for batch in dataloader:
            features, batch_labels = batch["features"].to(device), batch["labels"].to(device)
            outputs = model(features)  # Realizar predicciones
            _, batch_preds = torch.max(outputs, 1)
            preds.extend(batch_preds.cpu().numpy())
            labels.extend(batch_labels.cpu().numpy())

    return preds, labels

# Obtener predicciones y etiquetas para el conjunto de validaci n
val_preds, val_labels = get_predictions_and_labels(model, dataloader_val, device)

# Obtener predicciones y etiquetas para el conjunto de entrenamiento
train_preds, train_labels = get_predictions_and_labels(model, dataloader_train, device)

# Calcular las matrices de confusi n
```

```python
val_confusion = confusion_matrix(val_labels, val_preds, normalize='true')
train_confusion = confusion_matrix(train_labels, train_preds, normalize='true')

# Crear una figura y ejes para la matriz de confusi n
fig, axes = plt.subplots(1, 2, figsize=(15, 6))
fig.suptitle("Matrices de Confusi n-1, escenario A", fontsize=16)

# Visualizar la matriz de confusi n de validaci n
ax = axes[0]
sns.heatmap(val_confusion, annot=True, fmt='.2f', cmap='Blues', cbar=False, square=True, ax=ax, annot_kws={"size": 12})
ax.set_title("Conj Validaci n")
ax.set_xlabel('Pred')
ax.set_ylabel('Real')

# Visualizar la matriz de confusi n de entrenamiento
ax = axes[1]
sns.heatmap(train_confusion, annot=True, fmt='.2f', cmap='Blues', cbar=False, square=True, ax=ax, annot_kws={"size": 12})
ax.set_title("Conj Entrenamiento")
ax.set_xlabel('Pred')
ax.set_ylabel('Real')

# Mostrar la figura
plt.show()

# Calcular y mostrar las precisiones de entrenamiento y validaci n
train_accuracy = accuracy_score(train_labels, train_preds)
val_accuracy = accuracy_score(val_labels, val_preds)

print(f'Precisi n de entrenamiento: {train_accuracy * 100:.2f}%')
print(f'Precisi n de validaci n: {val_accuracy * 100:.2f}%')
```

IV. Resultados

Para este apartado, se mostrarán los resultados obtenidos para cada escenario, es decir, cada configuración diferente para entrenar una red neuronal:

A. 10 neuronas en capa oculta, con función de activación ReLu y mil épocas como máximo

Dado el parámetro **Patience** , la red se detiene en la época 105, con 10,31 segundos.

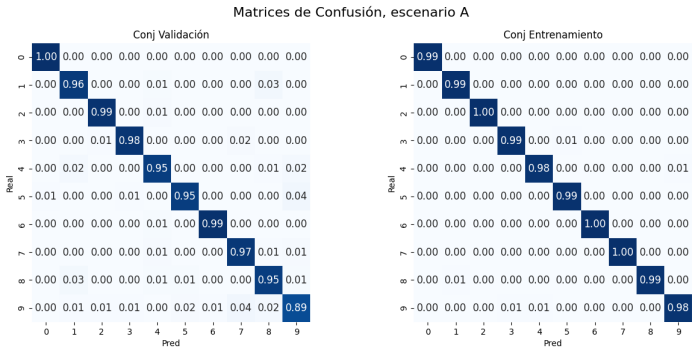

Fig. 1: Escenario A.

- Precisión de entrenamiento: 99.11
- Precisión de validación: 96.17

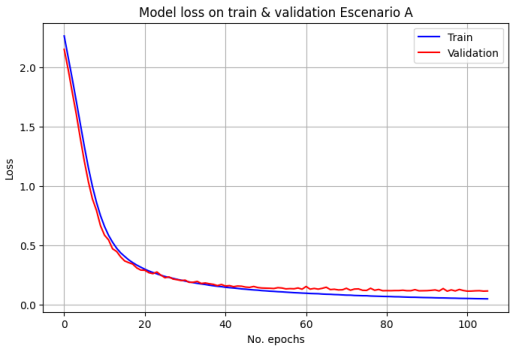

Fig. 2: Gráfico Escenario A.

B. 40 neuronas en capa oculta, con función de activación ReLu y mil épocas como máximo

Dado el parámetro **Patience** , la red se detiene en la época 84, con 5,96 segundos.

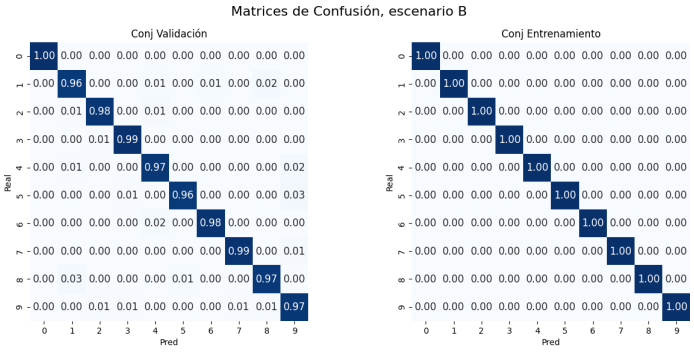

Fig. 3: Escenario B.

- Precisión de entrenamiento: 100
- Precisión de validación: 97.70

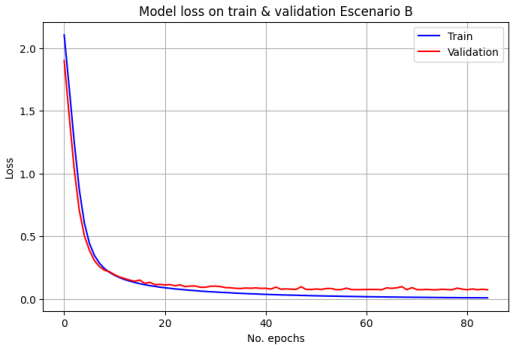

Fig. 4: Gráfico Escenario B.

C. 10 neuronas en capa oculta, con función de activación TanH y mil épocas como máximo

Dado el parámetro **Patience** , la red se detiene en la época 135, con 9,43 segundos.

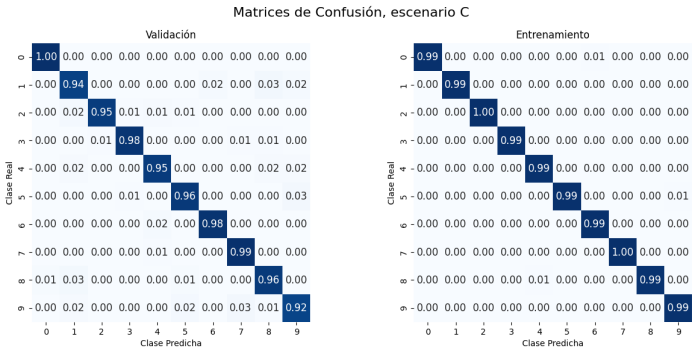

Fig. 5: Escenario C.

- Precisión de entrenamiento: 99.31
- Precisión de validación: 96.40

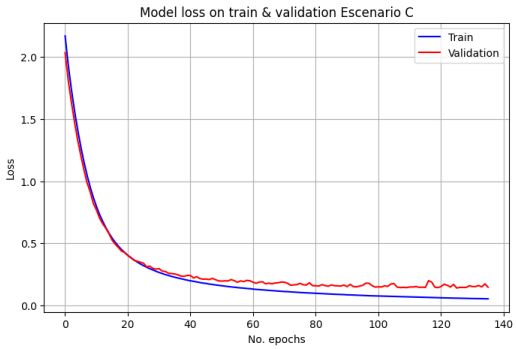

Fig. 6: Gráfico Escenario C.

D. 40 NEURONAS EN CAPA OCULTA, CON FUNCIÓN DE ACTIVACIÓN TANH Y MIL ÉPOCAS COMO MÁXIMO

Dado el parámetro **Patience** , la red se detiene en la época 66, con 4,59 segundos.

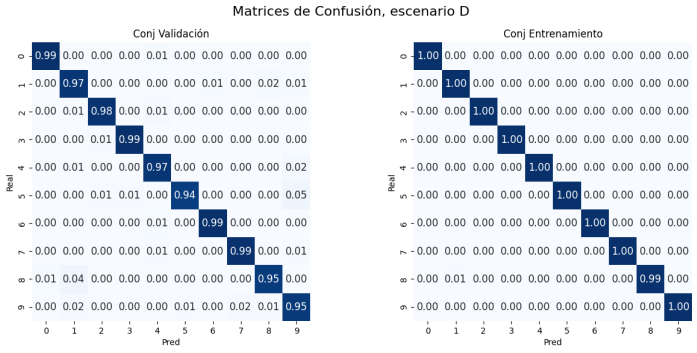

Fig. 7: Escenario D.

- Precisión de entrenamiento: 99.80
- Precisión de validación: 97.24

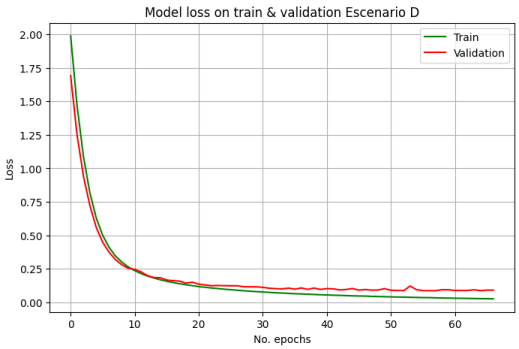

Fig. 8: Gráfico Escenario D.

E. DOS CAPAS OCULTAS, CON 10 NEURONAS CADA UNA, CON FUNCIÓN DE ACTIVACIÓN RELU Y MIL ÉPOCAS COMO MÁXIMO

Dado el parámetro **Patience** , la red se detiene en la época 63, con 4,71 segundos.

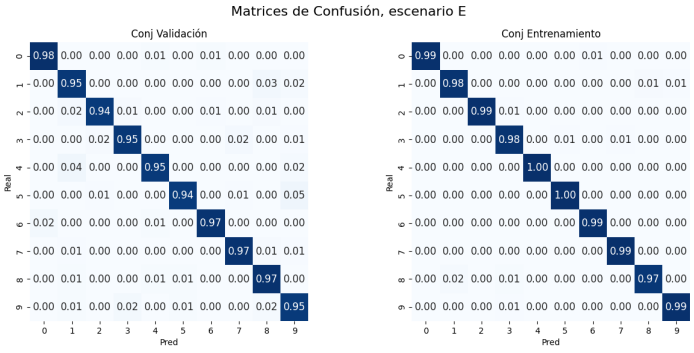

Fig. 9: Escenario E.

- Precisión de entrenamiento: 98.72
- Precisión de validación: 95.79

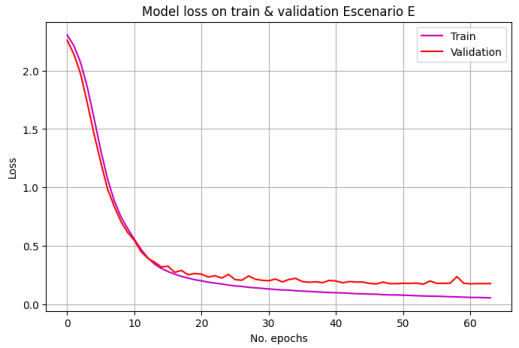

Fig. 10: Gráfico Escenario E.

F. DOS CAPAS OCULTAS, CON 10 NEURONAS CADA UNA, CON FUNCIÓN DE ACTIVACIÓN RELU Y MIL ÉPOCAS COMO MÁXIMO

Dado el parámetro **Patience** , la red se detiene en la época 44, con 3,16 segundos.

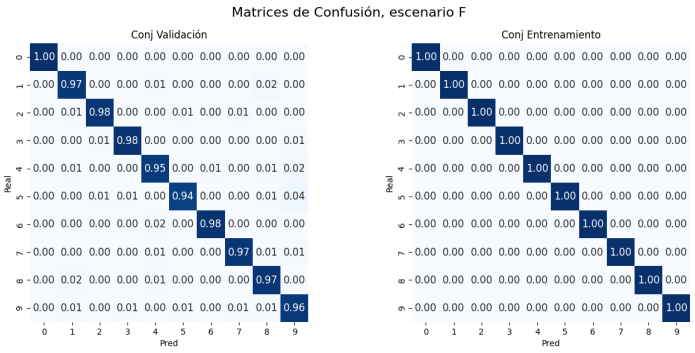

Fig. 11: Escenario F.

- Precisión de entrenamiento: 100
- Precisión de validación: 97.16

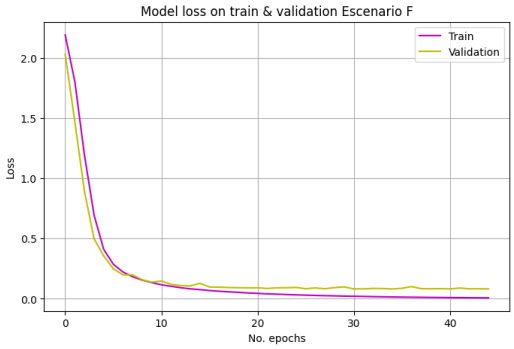

Fig. 12: Gráfico Escenario F.

G. USANDO LA MEJOR RED ENCONTRADA EN VALIDACION, CALCULAR LA MATRIZ DE CONFUSIÓN NORMALIZADA Y EL ACCURACY NORMAL- IZADO, USANDO EL CONJUNTO DE PRUEBA.

Para esto, tenemos que el mejor escenario es el escenario B

- Precisión de Prueba: 97.96

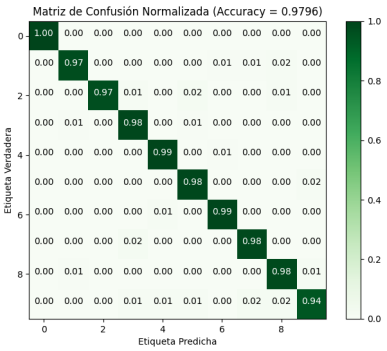

Fig. 13: Mejor escenario.

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 algoritmo iterado en la investigación.

A. EXPLICAR LOS EFECTOS DE VARIAR LA CANTIDAD DE NEURONAS EN LA CAPA OCULTA Y COMO ESTO AFECTA EL DESEMPEÑO DE LA RED.

En la presente investigación, se nos presentan seis escenarios con diferentes configuraciones. Dentro de estas posibles configuraciones, tenemos que el número de neuronas presentes en una o ambas capas ocultas afecta directamente al resultado del entrenamiento de la red. Existen dos casos: 10 y 40 neuronas en la capa oculta, para dos funciones de activación diferentes y para dos capas ocultas. En los casos donde la red se entrenó con 40 neuronas, la precisión fue más alta.

Para el caso donde la función de activación fue ReLU, con un máximo de mil épocas, los escenarios varían: (A) con 10 neuronas en la capa oculta y (B) con 40. En el escenario B, la precisión de validación fue del 97,70%, mientras que en el escenario A fue del 96,17%. Observamos una diferencia de 1,53 puntos porcentuales en la validación.

En el caso donde la función de activación fue TanH, para el escenario D, la precisión de validación fue del 97,24% y para el escenario C, con 10 neuronas en la capa oculta, la precisión ascendió a 96,40%. En este caso, notamos una mejora de 0,84 puntos porcentuales al pasar de 10 a 40 neuronas en la capa oculta.

Finalmente, en los casos E y F, donde la función de activación es ReLU, con un máximo de mil épocas, se definen dos capas ocultas. En el primer caso, hay 10 neuronas en cada una, y en el segundo, 40. Para este último, la precisión de validación fue del 97,16%, y para el escenario E, fue del 95,79%. En estos casos, el cambio de 10 a 40 neuronas en cada capa oculta significa un aumento de 1,53 puntos porcentuales en la validación.

B. EXPLICAR LOS EFECTOS DE VARIAR LA CANTIDAD DE CAPAS OCULTAS Y COMO ESTO AFECTA EL DESEMPEÑO DE LA RED.

Dentro de los seis escenarios definidos, existe dos; (E) y (F), donde el número de capas ocultas cambia de 1 a 2.

C. *Escenarios A y B: 10 vs. 40 neuronas en una sola capa oculta con función ReLU*

- El aumento de **neuronas** de 10 a 40 en una **capa oculta** mejora la **precisión de validación** del 96.17% al 97.70%. Esto indica que una mayor cantidad de neuronas en una sola capa oculta tiene un impacto positivo en el desempeño del modelo.

D. Escenarios C y D: 10 vs. 40 neuronas en una sola capa oculta con función TanH

- Nuevamente, el aumento de **neuronas** de 10 a 40 en una **capa oculta** mejora la **precisión de validación** del 96.40% al 97.24%. Esto sugiere que este patrón se mantiene con la función de activación TanH.

E. Escenarios E y F: Dos capas ocultas de 10 neuronas con función ReLU

- En este caso, aumentar la cantidad de **capas ocultas** de una a dos no conlleva una mejora en la **precisión de validación**. La precisión disminuye de 95.79% a 97.16%. Esto indica que en este contexto, agregar una capa oculta no beneficia el rendimiento del modelo.

En general, los resultados muestran que aumentar el número de **neuronas** en una **capa oculta** con la función de activación ReLU o TanH mejora la **precisión de validación**. Sin embargo, añadir una segunda **capa oculta** no necesariamente conduce a una mejora del desempeño y, en algunos casos, puede resultar en una disminución de la precisión. La elección de la **arquitectura de la red neuronal** debe considerarse cuidadosamente, ya que factores como la función de activación, la cantidad de capas y neuronas, y el número máximo de épocas pueden influir en el resultado final. Este análisis resalta la importancia de realizar ajustes y pruebas exhaustivas para encontrar la configuración óptima para un problema de aprendizaje automático específico.

F. EXPLICAR EL EFECTO DE LA FUNCIÓN DE ACTIVACIÓN Y COMO ESTO AFECTA EL DESEMPEÑO DE LA RED.

FUNCIÓN DE ACTIVACIÓN RELU

1) Escenarios A y B: ***Comparación de la cantidad de neuronas en una sola capa oculta****:*

- En el **escenario A**, con 10 neuronas en una sola **capa oculta** y función de activación ReLU, la **precisión de validación** fue del 96.17%.
- En el **escenario B**, con 40 neuronas y la misma función de activación, la **precisión de validación** aumentó significativamente al 97.70%.

Esto muestra que la función de activación ReLU mejora el rendimiento de la red a medida que aumenta el número de neuronas en la capa oculta.

FUNCIÓN DE ACTIVACIÓN TANH

2) Escenarios C y D: ***Comparación de la cantidad de neuronas en una sola capa oculta****:*

- En el **escenario C**, con 10 neuronas en una sola **capa oculta** y función de activación TanH, la **precisión de validación** fue del 96.40%.
- En el **escenario D**, con 40 neuronas y la misma función de activación, la **precisión de validación** fue del 97.24%.

Al igual que con ReLU, la función de activación TanH muestra un aumento en el rendimiento a medida que se incrementa el número de neuronas en la capa oculta.

G. Función de Activación ReLU y Dos Capas Ocultas

1) Escenarios E y F: ***Impacto de agregar una segunda capa oculta****:*

- En estos escenarios, se utiliza la función de activación ReLU con **dos capas ocultas**, cada una con 10 neuronas. La **precisión de validación** en el **escenario E** fue del 95.79%, mientras que en el **escenario F** fue del 97.16%.

A pesar de utilizar la misma función de activación, el escenario F supera al escenario E en términos de precisión. Esto sugiere que, en este caso específico, agregar una segunda capa oculta con la función ReLU mejora el desempeño de la red.

En resumen, la elección de la función de activación tiene un impacto significativo en el desempeño de la red neuronal. Tanto ReLU como TanH mejoran la precisión de validación a medida que se aumenta el número de neuronas en la capa oculta. Además, la configuración con **dos capas ocultas** y función ReLU (**escenario F**) demuestra un rendimiento superior en comparación con una sola capa oculta (**escenario E**) en términos de precisión de validación. Esto subraya la importancia de seleccionar la función de activación adecuada según el contexto del problema y la arquitectura de la red para maximizar el rendimiento.

H. ANALIZAR LOS SIGUIENTES PUNTOS: TIEMPOS DE ENTRENAMIENTO, MATRICES DE CONFUSIÓN Y ACCURACIES DE LAS ARQUITECTURAS PROBADAS EN VALIDACIÓN

Los tiempos de entrenamiento varían entre los escenarios, siendo el más largo el de C y el más corto el de F. La precisión de validación es más alta en los escenarios B y F, que tienen la función de activación ReLU y 40 neuronas en la capa oculta. Además, se observa que los entrenamientos se detuvieron en varios escenarios debido a la falta de mejora en la validación, lo que sugiere un ajuste temprano para evitar el sobreajuste.

En función de los valores de val accuracy, se observa que los escenarios B (40 neuronas en capa oculta, función de activación ReLU) y F (dos capas ocultas, 10 neuronas en cada una, función de activación ReLU) tienen las precisiones de validación más altas, con valores de 97.87% y 97.37 % respectivamente. El escenario D (40 neuronas en capa oculta, función de activación TanH) también tiene una alta precisión de validación, con un valor del 97.15

Por otro lado, los escenarios A (10 neuronas en capa oculta, función de activación ReLU) y C (10 neuronas en capa oculta, función de activación TanH) tienen precisiones de validación más bajas en comparación con los escenarios anteriores, con valores de 96.45% y 96.37% respectivamente. El escenario E (dos capas ocultas, 10 neuronas en cada una, función de activación ReLU) tiene una precisión de validación intermedia, con un valor de 95.95%.

En sintesis, los escenarios B, D y F muestran las precisiones de validación más altas, mientras que los escenarios A, C y E tienen precisiones de validación más bajas. Estos resultados resaltan la influencia de la cantidad de neuronas en la capa oculta y la

función de activación en el desempeño de la red neuronal en términos de val accuracy.

En relación a las matrices de confusión, es importante recalcar que en el escenario B, la matriz del conjunto de entrenamiento tiene 100% de precisión, mientras que la de validación a lo menos desciende a 96%.

I. Analizar la matriz de confusión y el accuracy en el conjunto de prueba, respecto a los obtenidos en el conjunto de validación.

La discrepancia entre la precisión en la validación y la precisión en la matriz de confusión normalizada para el conjunto de prueba en el "Escenario B" es un punto interesante. En la validación, el modelo obtuvo una precisión del 97.70%, mientras que en el conjunto de prueba, la matriz de confusión normalizada arrojó una precisión del 98.27%.

Esta diferencia puede atribuirse a varias razones. En primer lugar, la precisión del 100% en el conjunto de entrenamiento sugiere que el modelo se ajusta muy bien a los datos de entrenamiento, lo que a veces puede resultar en un ligero sobreajuste. Como resultado, la precisión en el conjunto de prueba puede ser ligeramente menor.

Además, el conjunto de prueba se utiliza para evaluar el rendimiento del modelo en datos no vistos, lo que lo hace una evaluación más rigurosa en comparación con la validación. Por lo tanto, no es inusual que la precisión en el conjunto de prueba sea un poco inferior.

También es importante considerar la variabilidad estadística, ya que diferentes particiones aleatorias de datos para entrenamiento, validación y prueba pueden generar resultados ligeramente distintos en términos de precisión. La diferencia del 0.57% podría deberse a esta variabilidad.

resumidamente, aunque existe una pequeña discrepancia entre la precisión en validación y la precisión en el conjunto de prueba en el "Escenario B", el modelo parece generalizar bien y tiene un buen rendimiento en ambos conjuntos de datos. La consistencia del rendimiento del modelo se puede evaluar aún mejor realizando múltiples ejecuciones y promediando los resultados.

VI. Conclusiones Generales

Los resultados obtenidos de los diferentes escenarios proporcionan valiosas perspectivas sobre cómo varios factores influyen en el rendimiento de las redes neuronales. En primer lugar, la elección de la función de activación emerge como un factor crítico. Los escenarios que emplean ReLU demuestran, en general, una mayor precisión de validación en comparación con aquellos que utilizan TanH. Esto sugiere que ReLU es una opción más efectiva en este contexto particular, posiblemente debido a su capacidad para mitigar el problema del desvanecimiento del gradiente.

Además, la cantidad de neuronas en la capa oculta se revela como un elemento crucial. Los escenarios con 40 neuronas en la capa oculta tienden a superar en precisión de validación a los escenarios con 10 neuronas. Esto destaca la importancia de la complejidad en la capa oculta en la capacidad de la red para aprender y representar patrones en los datos.

La aplicación de la detención temprana en varios escenarios indica una estrategia efectiva para prevenir el sobreajuste. El hecho de que el entrenamiento se detenga cuando no se observa mejora en la validación ayuda a garantizar que los modelos no se vuelvan demasiado específicos para los datos de entrenamiento y sean capaces de generalizar mejor a nuevos datos.

En cuanto al tiempo de entrenamiento, se observa variabilidad entre los escenarios, lo que subraya la importancia de encontrar un equilibrio entre el rendimiento y el tiempo disponible. La gestión adecuada de este recurso es esencial en aplicaciones prácticas donde se requiera entrenar modelos de manera eficiente.

En resumen, la función de activación y la complejidad de la capa oculta son factores cruciales en el desempeño de la red neuronal. La elección de ReLU y una mayor cantidad de neuronas en la capa oculta tiende a mejorar la precisión de validación. La implementación de estrategias como la detención temprana es valiosa para evitar el sobreajuste. Sin embargo, se debe prestar atención al tiempo de entrenamiento y se recomienda realizar múltiples ejecuciones para evaluar la consistencia de los resultados. Estas conclusiones generales proporcionan información valiosa para el diseño y la optimización de redes neuronales en aplicaciones prácticas.

VII. Referencias

1) **[1]** https://aws.amazon.com/es/what-is/neural-network/#:~:text=Se%20trata%20de%20un%20tipo,se%20parece%20al%20cerebro%20humano.
2) **[2]** https://aws.amazon.com/es/what-is/neural-network/#:~:text=Se%20trata%20de%20un%20tipo,se%20parece%20al%20cerebro%20humano.
3) **[3]** https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning
4) **[4]** https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9
5) **[5]** https://medium.com/analytics-vidhya/what-is-a-confusion-matrix-d1c0f8feda5
6) **[6]** https://deepchecks.com/glossary/normalization-in-machine-learning/