Tarea N°2:

Redes Neuronales

Rodrigo Aníbal Llancao Zúñiga
Escuela de Ingeniería, Universidad de O'Higgins
28, Octubre, 2023

***Abstract*—El objetivo consistió en entrenar y validar redes neuronales con diferentes configuraciones arquitectónicas, incluyendo variaciones en el numero de capas ocultas, el numero de neuronas y las funciones de activación. Se utilizó el conjunto de datos Optical Recognition of Handwritten Digits, que consta de 64 caracterısticas, 10 clases y un total de 5620 muestras. A través del uso de PyTorch, se implementaron y entrenaron los clasificadores de dígitos, analizando las perdidas de entrenamiento y validación, generando matrices de confusión normalizadas y evaluando la precisión en los conjuntos de entrenamiento, validación y prueba. Los resultados obtenidos permitieron examinar el impacto de las variaciones arquitectónicas en el rendimiento de las redes, brindando así información valiosa sobre la importancia de seleccionar configuraciones adecuadas para la clasificación de dígitos manuscritos.**

I. Introducción

El presente informe se centra en el desarrollo y evaluación de un modelo de red neuronal para la clasificación de datos. Se emplea una red neuronal implementada con PyTorch para abordar un problema de clasificación sobre un conjunto de datos. El objetivo primordial es comprender y analizar el proceso de entrenamiento de una red neuronal, así como evaluar su desempeño en conjuntos de datos de entrenamiento y validación.

En este informe, se detalla el proceso de entrenamiento del modelo, incluyendo la preparación de los datos, la configuración de la red neuronal, el ajuste de hiperparámetros y la evaluación del modelo en términos de pérdida (loss) y precisión (accuracy) tanto en los datos de entrenamiento como en los de validación.

Además, se exploran herramientas como la visualización de la evolución del loss a lo largo de las épocas, y se analiza la matriz de confusión para comprender el desempeño del modelo en la clasificación de diferentes clases.

El informe se estructura en secciones que abarcan desde la preparación de datos hasta la evaluación y visualización de resultados, proporcionando una visión general del proceso y los resultados obtenidos en este estudio de clasificación con redes neuronales.

II. Marco Teórico

A. Conjunto de datos Optical Recognition of Handwritten Digits Data Set:

Conjunto de datos Optical Recognition of Handwritten Digits Data Set: El conjunto de datos Optical Recognition of Handwritten Digits Data Set es ampliamente utilizado en el campo del reconocimiento de dígitos manuscritos. Fue creado por E.Alpaydin y F. Alimoglu en el Laboratorio de Visión por computadora del Instituto de Tecnología de Bogazici en Estambul, Turquía. Este conjunto de datos consiste en un total de 5620 imágenes de dígitos escritos a mano. Cada imagen tiene un tamaño de 8x8 píxeles, lo que da un total de 64 características por muestra. Cada p´ıxel puede tener un valor entre 0 y 16, que representa el nivel de gris correspondiente al tono de la tinta utilizada en el dígito. Además, cada muestra está etiquetada con la clase correcta que representa el dígito correspondiente. El conjunto de datos se divide en dos partes: un conjunto de entrenamiento y un conjunto de prueba. El conjunto de entrenamiento contiene 3823 muestras, mientras que el conjunto de prueba contiene 1797 muestras. Esta división permite evaluar el desempeño del modelo entrenado en datos previamente no observados.

B. Redes Neuronales

Las redes neuronales son modelos computacionales inspirados en el funcionamiento del cerebro humano. Consisten en un conjunto de unidades llamadas neuronas que están interconectadas entre sí. Las redes neuronales se componen de una capa de entrada, una o más capas ocultas y una capa de salida. Cada capa está formada por un conjunto de neuronas que procesan y transmiten información.

C. GPU de Google Colab

Google Colab proporciona acceso gratuito a GPU (Unidad de Procesamiento Gráfico) para acelerar el entrenamiento de redes neuronales. Utilizar la GPU en Google Colab permite procesar cálculos intensivos de manera más eficiente y reduce significativamente el tiempo de entrenamiento de los modelos.

D. PyTorch

PyTorch es un framework de aprendizaje automático de código abierto basado en Python. Proporciona una amplia gama de herramientas y funciones para la construcción y entrenamiento de modelos de redes neuronales. PyTorch facilita la implementación de algoritmos de aprendizaje profundo y ofrece una interfaz sencilla y flexible para trabajar con tensores y operaciones en redes neuronales.

E. torch.nn

El módulo `torch.nn` de PyTorch proporciona las herramientas y clases necesarias para construir redes neuronales. Contiene implementaciones de diferentes capas, funciones de activación, funciones de pérdida y optimizadores que se utilizan para definir y entrenar modelos de redes neuronales.

F. Función de pérdida: Entropía cruzada

La entropía cruzada, también conocida como *cross-entropy*, es una función de pérdida comúnmente utilizada en problemas de clasificación. Se utiliza para medir la diferencia entre la distribución de probabilidad predicha por el modelo y la distribución de probabilidad real de los datos de entrenamiento. La entropía cruzada penaliza las predicciones incorrectas, fomentando así el aprendizaje preciso del modelo.

G. Optimizador Adam

Adam es un algoritmo de optimización ampliamente utilizado para ajustar los pesos de las redes neuronales durante el entrenamiento. Se basa en el método del gradiente descendente estocástico y adapta automáticamente la tasa de aprendizaje para cada parámetro del modelo. El optimizador Adam es eficiente y proporciona buenos resultados en la mayoría de los casos de entrenamiento.

H. Matriz de confusión

La matriz de confusión es una herramienta utilizada para evaluar el rendimiento de un modelo de clasificación. Muestra la cantidad de muestras clasificadas correctamente e incorrectamente para cada clase. La matriz de confusión es especialmente útil para identificar los errores de clasificación y comprender cómo el modelo se confunde entre las diferentes clases.

I. Accuracy

La exactitud (*accuracy*) es una medida utilizada para evaluar el rendimiento de un modelo de clasificación. Se calcula dividiendo el número de predicciones correctas entre el número total de muestras. La exactitud indica qué tan bien el modelo es capaz de clasificar correctamente las muestras en comparación con el número total de muestras.

III. Metodología

En esta sección se describe la metodología utilizada para abordar el problema de clasificación de dígitos manuscritos mediante redes neuronales. El enfoque adoptado se basa en las indicaciones proporcionadas en el enunciado de la tarea y en el código base entregado.

A. Carga de datos

Se comienza por cargar el conjunto de datos utilizado, conocido como Optical Recognition of Handwritten Digits Data Set. Los datos se encuentran divididos en conjuntos de entrenamiento, validación y prueba. Para su manipulación y procesamiento, se emplea la librería PyTorch, la cual proporciona las herramientas necesarias para trabajar con redes neuronales.

B. Definición de la arquitectura de la red neuronal

Se procede a definir diferentes arquitecturas de redes neuronales, siguiendo las especificaciones establecidas en el enunciado de la tarea. Cada red neuronal está compuesta por una capa de entrada con una dimensionalidad de 64, una o dos capas ocultas con un número variable de neuronas y una capa de salida con 10 neuronas, donde se aplica una función de activación *softmax*. La biblioteca `torch.nn` de PyTorch es utilizada para la construcción de la red neuronal. A continuación se muestra un ejemplo del primer modelo, los códigos de los otros modelos serán ignorados para una mejor lectura del informe.

Código 1: Creación del modelo

```
model = nn.Sequential(
          nn.Linear(64, 10),
          nn.ReLU(),
          nn.Linear(10,10)
        )

device = torch.device('cuda')

model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

C. Entrenamiento de la red neuronal

Una vez definidas las arquitecturas de las redes neuronales, se procede al entrenamiento de cada una de ellas utilizando el conjunto de entrenamiento. Durante el proceso de entrenamiento, se emplea la función de pérdida de entropía cruzada como medida de la discrepancia entre las salidas de la red y las etiquetas reales de los datos. El algoritmo de optimización Adam es utilizado para ajustar los parámetros de la red y minimizar la función de pérdida. Se establece un límite de 1000 épocas como máximo para el entrenamiento. Luego, para evitar el sobreajuste de la red, se implementa una funcionalidad que detiene el entrenamiento cuando el valor de la pérdida de validación comienza a aumentar, mientras que la pérdida de entrenamiento sigue disminuyendo la cual se muestra a continuación:

Código 2: Implementación del Early Stopping

```
if epoch > 0:
    if ((loss_val[epoch] > loss_val[epoch-1]) and (
     loss_train[epoch] < loss_train[epoch-1])):
        patience +=1
        best_model = model.state_dict()
        # Comprobar si se debe detener el entrenamiento
    if patience == 250:
        best_model = model.state_dict()
        print("Early stopping. Validation loss hasn't
     improved for {} epochs.".format(patience))
        break  # Detener el bucle de entrenamiento
```

Accedemos al valor del loss_val en la época presente y preguntamos si es mayor al anterior, luego lo mismo con el loss_train pero preguntamos si es menor que el de la época anterior para así compararlos mediante la operación lógica AND y saber si hay overfitting o no. Declaramos un límite llamado patience (el cual fue inicializado en 0) el cual será el encargado de detener el entrenamiento. Además se guarda el estado del modelo en la última instancia calculada. A

continuación, el código completo de dos redes entrenadas a lo largo de la tarea que serán analizados más tarde:

Código 3: Red neuronal con 10 neuronas en la capa oculta y función de activación Tanh

```
# Crear modelo
model_c = nn.Sequential(
    nn.Linear(64, 10),
    nn.Tanh(),
    nn.Linear(10, 10)
)
model_c = model_c.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_c.parameters(), lr=1e-3)

# Inicializar variables de tiempo
start = time.time()

# Guardar resultados del loss y epocas que dur  el entrenamiento
loss_train_c = []
loss_val_c = []
epochs_c = []

best_train_loss = 0.0  # Inicializar
patience = 0  # N mero de  pocas  sin mejora antes de detener el entrenamiento
best_model_c= None
# Entrenamiento de la red por n epocas
for epoch in range(1000):

  # Guardar loss de cada batch
  loss_train_batches_c = []
  loss_val_batches_c = []

  # Entrenamiento
  ----------

  model_c.train()
  # Debemos recorrer cada batch (lote de los datos)
  for i, data in enumerate(dataloader_train, 0):
    # Procesar batch actual
    inputs = data["features"].to(device) # Caracter sticas
    labels = data["labels"].to(device)   # Clases
    # zero the parameter gradients
    optimizer.zero_grad()
    # forward + backward + optimize
    outputs = model_c(inputs)            # Predicciones
    loss_c = criterion(outputs, labels) # Loss de entrenamiento
    loss_c.backward()                    # Backpropagation
    optimizer.step()

    # Guardamos la p rdida de entrenamiento en el batch actual
    loss_train_batches_c.append(loss_c.item())

  # Guardamos el loss de entrenamiento de la  poca  actual
  loss_train_c.append(np.mean(loss_train_batches_c)) # Loss promedio de los batches

  # Predicci n en conjunto de validaci n
  ---------------------------------------
  model_c.eval()
  with torch.no_grad():
    # Iteramos dataloader_val para evaluar el modelo en los datos de validaci n
    for i, data in enumerate(dataloader_val, 0):
      # Procesar batch actual
      inputs = data["features"].to(device) # Caracter sticas
      labels = data["labels"].to(device)   # Clases

      outputs = model_c(inputs)              # Obtenemos predicciones

      # Guardamos la p rdida de validaci n en el batch actual
      loss_c = criterion(outputs, labels)
      loss_val_batches_c.append(loss_c.item())


  # Guardamos el Loss de validaci n de la  poca  actual
  loss_val_c.append(np.mean(loss_val_batches_c)) # Loss promedio de los batches

  # Guardamos la  poca
  epochs_c.append(epoch)

  # Imprimir la p rdida de entrenamiento/validaci n en la  poca  actual
  print(("Epoch: %d, train loss: %.4f, val loss: %.4f"  %(epoch, loss_train_c[epoch], loss_val_c[epoch])))

  if epoch > 0:
    if (loss_val_c[epoch] > loss_val_c[epoch-1] and loss_train_c[epoch] < loss_train_c[epoch-1]):
        patience +=1
        # Comprobar si se debe detener el entrenamiento
    if patience == 200:
        best_model_c = model_c.state_dict()
        print("Early stopping. Validation loss hasn't improved for {} epochs.".format(patience))
        break  # Detener el bucle de entrenamiento

end = time.time()
print('Finished Training, total time %f seconds' % (end - start))

# Graficar el loss de entrenamiento y validaci n en funci n del tiempo
plt.figure()
plt.plot(epochs_c, loss_train_c, label="Train Loss", linewidth=2)
plt.plot(epochs_c, loss_val_c, label="Validation Loss", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("Loss vs. Epoch")
plt.show()



# Obtener las predicciones en el conjunto de entrenamiento
model_c.load_state_dict(best_model_c)
model_c.eval()
train_predictions_c = []
train_labels_c = []

with torch.no_grad():
    for data in dataloader_train:
        inputs = data["features"].to(device)
        labels = data["labels"].to(device)
        outputs = model_c(inputs)
        _, predicted = torch.max(outputs.data, 1)
        train_predictions_c.extend(predicted.tolist())
        train_labels_c.extend(labels.tolist())

# Calcular la matriz de confusi n
confusion_c = confusion_matrix(train_labels_c, train_predictions_c, normalize='true')

# Calcular el accuracy
accuracy_c = accuracy_score(train_labels_c, train_predictions_c)

# Mostrar la matriz de confusi n con colores
disp = ConfusionMatrixDisplay(confusion_c, display_labels=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
disp.plot(cmap=plt.cm.Blues, values_format=".2f")
plt.title("Normalized Confusion Matrix (Training)")
plt.show()

print("Normalized Accuracy (Training):", accuracy_c)

# Obtener las predicciones en el conjunto de validaci n
model_c.eval()
val_predictions_c = []
val_labels_c = []

with torch.no_grad():
    for data in dataloader_val:
        inputs = data["features"].to(device)
        labels = data["labels"].to(device)
        outputs = model_c(inputs)
        _, predicted = torch.max(outputs.data, 1)
        val_predictions_c.extend(predicted.tolist())
        val_labels_c.extend(labels.tolist())
```

```

# Calcular la matriz de confusi n
val_confusion_c = confusion_matrix(val_labels_c,
    val_predictions_c, normalize='true')

# Calcular el accuracy
val_accuracy_c = accuracy_score(val_labels_c,
    val_predictions_c)

# Mostrar la matriz de confusi n con colores
val_disp_c = ConfusionMatrixDisplay(val_confusion_c,
    display_labels=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
val_disp_c.plot(cmap=plt.cm.Blues, values_format=".2f")
plt.title("Normalized Confusion Matrix (Validation)")
plt.show()

print("Normalized Accuracy (Validation):", val_accuracy_c)
```

Código 4: Red neuronal con 2 capas ocultas

```
# Crear modelo
model_f = nn.Sequential(
    nn.Linear(64, 40),
    nn.ReLU(),
    nn.Linear(40, 40),
    nn.ReLU(),
    nn.Linear(40, 10)
)
model_f = model_f.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_f.parameters(), lr=1e-3)

# Inicializar variables de tiempo
start = time.time()

# Guardar resultados del loss y epocas que dur  el
    entrenamiento
loss_train_f = []
loss_val_f = []
epochs_f = []

best_train_loss = 0.0  # Inicializar
patience = 0  # N mero de  pocas  sin mejora antes de
    detener el entrenamiento
best_model_f = None
# Entrenamiento de la red por n epocas
for epoch in range(1000):

  # Guardar loss de cada batch
  loss_train_batches_f = []
  loss_val_batches_f = []

  # Entrenamiento
  ----------------------------------------------------

  model_f.train()
  # Debemos recorrer cada batch (lote de los datos)
  for i, data in enumerate(dataloader_train, 0):
    # Procesar batch actual
    inputs = data["features"].to(device) # Caracter sticas
    labels = data["labels"].to(device)   # Clases
    # zero the parameter gradients
    optimizer.zero_grad()
    # forward + backward + optimize
    outputs = model_f(inputs)           # Predicciones
    loss_f = criterion(outputs, labels) # Loss de
     entrenamiento
    loss_f.backward()                   # Backpropagation
    optimizer.step()

    # Guardamos la p rdida de entrenamiento en el batch
     actual
    loss_train_batches_f.append(loss_f.item())

  # Guardamos el loss de entrenamiento de la  poca  actual
  loss_train_f.append(np.mean(loss_train_batches_f)) # Loss
     promedio de los batches

  # Predicci n en conjunto de validaci n
  ------------------------------------
  model_f.eval()
  with torch.no_grad():
    # Iteramos dataloader_val para evaluar el modelo en los
     datos de validaci n
    for i, data in enumerate(dataloader_val, 0):
      # Procesar batch actual
      inputs = data["features"].to(device) #
     Caracter sticas
      labels = data["labels"].to(device)   # Clases

      outputs = model_f(inputs)              # Obtenemos
     predicciones

      # Guardamos la p rdida de validaci n en el batch
     actual
      loss_f = criterion(outputs, labels)
      loss_val_batches_f.append(loss_f.item())


  # Guardamos el Loss de validaci n de la  poca  actual
  loss_val_f.append(np.mean(loss_val_batches_f)) # Loss
     promedio de los batches

  # Guardamos la  poca
  epochs_f.append(epoch)

  # Imprimir la p rdida de entrenamiento/validaci n en la
      poca  actual
  print(("Epoch: %d, train loss: %.4f, val loss: %.4f"  %(
     epoch, loss_train_f[epoch], loss_val_f[epoch])))

  # Tenemos el loss de entrenamiento y validaci n,  Como
     ser a el early-stopping?
  if epoch > 0:
    if (loss_val_f[epoch] > loss_val_f[epoch-1] and
     loss_train_f[epoch] < loss_train_f[epoch-1]):
        patience +=1
        # Comprobar si se debe detener el entrenamiento
    if patience == 100:
        best_model_f = model_f.state_dict()
        print("Early stopping. Validation loss hasn't
     improved for {} epochs.".format(patience))
        break  # Detener el bucle de entrenamiento

end = time.time()
print('Finished Training, total time %f seconds' % (end -
    start))

# Graficar el loss de entrenamiento y validaci n en
    funci n del tiempo
plt.figure()
plt.plot(epochs_f, loss_train_f, label="Train Loss",
    linewidth=2)
plt.plot(epochs_f, loss_val_f, label="Validation Loss",
    linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("Loss vs. Epoch")
plt.show()



# Obtener las predicciones en el conjunto de entrenamiento
model_f.load_state_dict(best_model_f)
model_f.eval()
train_predictions_f = []
train_labels_f = []

with torch.no_grad():
    for data in dataloader_train:
        inputs = data["features"].to(device)
        labels = data["labels"].to(device)
        outputs = model_f(inputs)
        _, predicted = torch.max(outputs.data, 1)
        train_predictions_f.extend(predicted.cpu().numpy())
        train_labels_f.extend(labels.cpu().numpy())

# Calcular la matriz de confusi n
confusion_f = confusion_matrix(train_labels_f,
    train_predictions_f, normalize='true')

# Calcular el accuracy
accuracy_f = accuracy_score(train_labels_f,
    train_predictions_f)

# Mostrar la matriz de confusi n con colores
disp = ConfusionMatrixDisplay(confusion_f, display_labels
    =[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
disp.plot(cmap=plt.cm.Blues, values_format=".2f")
plt.title("Normalized Confusion Matrix (Training)")
```

```
plt.show()

print("Normalized Accuracy (Training):", accuracy_f)

# Obtener las predicciones en el conjunto de validaci n
model_f.eval()
val_predictions_f = []
val_labels_f = []

with torch.no_grad():
    for data in dataloader_val:
        inputs = data["features"].to(device)
        labels = data["labels"].to(device)
        outputs = model_f(inputs)
        _, predicted = torch.max(outputs.data, 1)
        val_predictions_f.extend(predicted.cpu().numpy())
        val_labels_f.extend(labels.cpu().numpy())

# Calcular la matriz de confusi n
val_confusion_f = confusion_matrix(val_labels_f,
    val_predictions_f, normalize='true')

# Calcular el accuracy
val_accuracy_f = accuracy_score(val_labels_f,
    val_predictions_f)

# Mostrar la matriz de confusi n con colores
val_disp_f = ConfusionMatrixDisplay(val_confusion_f,
    display_labels=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
val_disp_f.plot(cmap=plt.cm.Blues, values_format=".2f")
plt.title("Normalized Confusion Matrix (Validation)")
plt.show()

print("Normalized Accuracy (Validation):", val_accuracy_f)
```

D. Evaluación del desempeño de la red neuronal

Una vez finalizado el entrenamiento, se evalúa el desempeño de cada red neuronal mediante la obtención de métricas relevantes. Se calcula la pérdida de entrenamiento y de validación, así como el tiempo total requerido para el entrenamiento de cada red. Estos valores permiten tener una idea del desempeño y la eficiencia de cada arquitectura de red neuronal. Además, se grafican las curvas de pérdida de entrenamiento y de validación en función del tiempo, lo cual brinda una visualización del proceso de entrenamiento y su convergencia. Quedado de la forma:

```
Epoch: 242, train loss: 0.0088, val loss: 0.2412
Epoch: 243, train loss: 0.0087, val loss: 0.1777
Epoch: 244, train loss: 0.0086, val loss: 0.1955
Epoch: 245, train loss: 0.0084, val loss: 0.2418
Epoch: 246, train loss: 0.0083, val loss: 0.1810
Epoch: 247, train loss: 0.0082, val loss: 0.1801
Epoch: 248, train loss: 0.0082, val loss: 0.1796
Epoch: 249, train loss: 0.0081, val loss: 0.1843
110
Finished Training, total time 15.502115 seconds
```

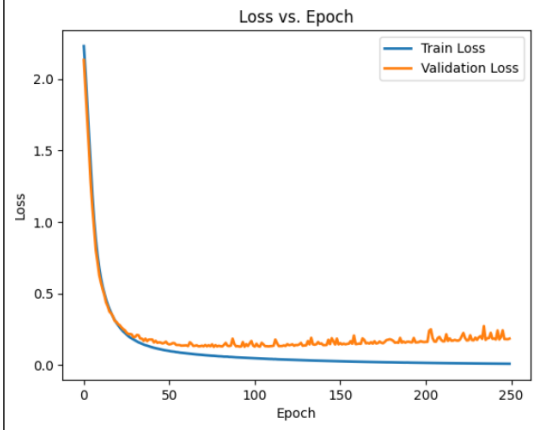

Fig. 1: Exposición de resultados y gráfica del entrenamiento preliminar

E. Selección de la mejor red neuronal

Una vez evaluadas todas las redes neuronales, se selecciona aquella que haya obtenido el mayor accuracy en el conjunto de validación como la mejor red encontrada. Esta red será utilizada en el siguiente paso para evaluar su desempeño en el conjunto de prueba.

Código 5: Selección mejor red neuronal

```
accuracies = [val_accuracy_a, val_accuracy_b,
    val_accuracy_c, val_accuracy_d, val_accuracy_e,
    val_accuracy_f]
model_names = ["Model A", "Model B", "Model C", "Model D",
    "Model E", "Model F"]

plt.bar(model_names, accuracies)
plt.xlabel('Modelos')
plt.ylabel('Accuracy')
plt.title('Comparacion Accuracy de los Modelos en
    Validacion')
plt.ylim(0, 1)
plt.xticks(rotation=45)

for i, acc in enumerate(accuracies):
    plt.text(i, acc, f'{acc:.2f}', ha='center', va='bottom'
    )

plt.show()
```

F. Evaluación en el conjunto de prueba

Utilizando la mejor red neuronal seleccionada previamente, se realiza una evaluación en el conjunto de prueba. Se calcula la matriz de confusión normalizada y el accuracy normalizado utilizando este conjunto de datos. Esto permite obtener una medida del desempeño de la red neuronal en datos no vistos previamente, proporcionando una evaluación más completa de su capacidad de generalización.

Código 6: Calculando las métricas para el mejor modelo con los datos de prueba

```
# Cargar el estado de la mejor red en el modelo
model_f.load_state_dict(best_model_f)

# Evaluar la mejor red en el conjunto de prueba
model_f.eval()
pred_test_f = []
true_test_f = []

with torch.no_grad():
    for data_test in dataloader_test:
        inputs_test = data_test["features"].to(device)
        labels_test = data_test["labels"].to(device)
        outputs_test = model_f(inputs_test)
        _, predicted_test = torch.max(outputs_test.data, 1)

        pred_test_f.extend(predicted_test.tolist())
        true_test_f.extend(labels_test.tolist())

confusion_matrix_test_f = confusion_matrix(true_test_f,
    pred_test_f, normalize='true')
accuracy_test_f = accuracy_score(true_test_f, pred_test_f)

disp_test_f = ConfusionMatrixDisplay(confusion_matrix=
    confusion_matrix_test_f, display_labels=range(10))
disp_test_f.plot(cmap='Blues')
plt.title('Confusion Matrix - Test (Case f)')
plt.show()

print(f"Accuracy - Test (Case f): {accuracy_test_f}")
```

G. Análisis de los resultados

Por último, se lleva a cabo un análisis detallado de los resultados obtenidos. Se exploran las variaciones en la cantidad de neuronas en la capa oculta, el número de capas

ocultas y las funciones de activación empleadas. Se comparan los tiempos de entrenamiento, las matrices de confusión y las precisiones de las distintas arquitecturas evaluadas en el conjunto de validación. Asimismo, se examina la matriz de confusión y la precisión en el conjunto de pruebas en relación con los obtenidos en el conjunto de validación.

Código 7: Código de línea de los accuracies de entrenamiento y validación

```
# Gr fico de l nea de los accuracies de entrenamiento y
    validaci n
models = ['a', 'b', 'c', 'd', 'e', 'f']
accuracy_train = [accuracy_a, accuracy_b, accuracy_c,
    accuracy_d, accuracy_e, accuracy_f]
accuracy_val = [val_accuracy_a, val_accuracy_b,
    val_accuracy_c, val_accuracy_d, val_accuracy_e,
    val_accuracy_f]

plt.plot(models, accuracy_train, label='Training Accuracy',
     marker='o')
plt.plot(models, accuracy_val, label='Validation Accuracy',
     marker='o')
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Accuracy Comparison - Training vs Validation')
plt.legend()
plt.show()
```

IV. Resultados

A continuación, se presentan los principales resultados obtenidos por cada uno de los modelos, siguiendo los requisitos establecidos en el enunciado.

A. Loss

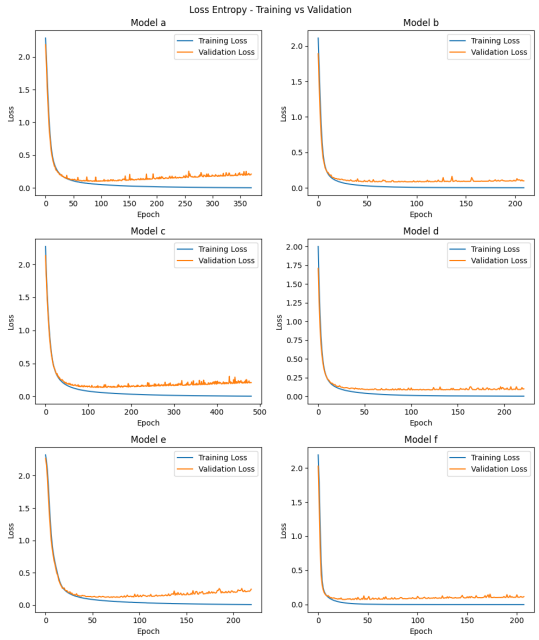

Fig. 2: Gráfico de la pérdida de entrenamiento y validación para cada red entrenada

B. Accuracy

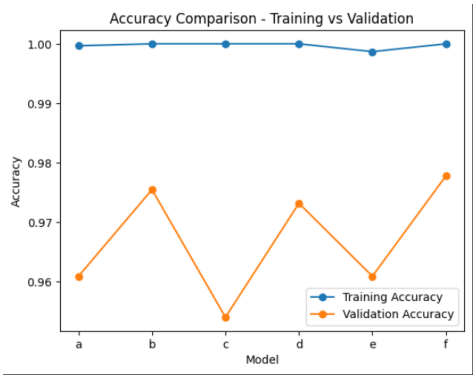

Fig. 3: Gráfico de línea de las precisiones de entrenamiento y validación

C. Tiempo

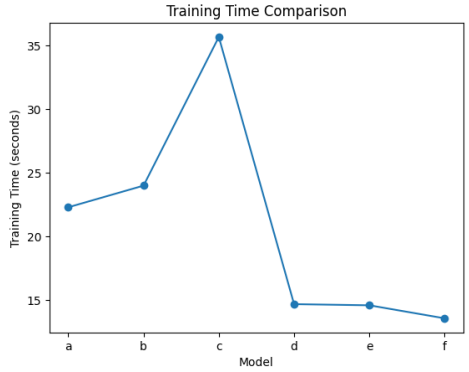

Fig. 4: Gráfico de línea para comparar los tiempos de ejecución de cada red neuronal entrenada

D. Matrices de Confusión - Entrenamiento

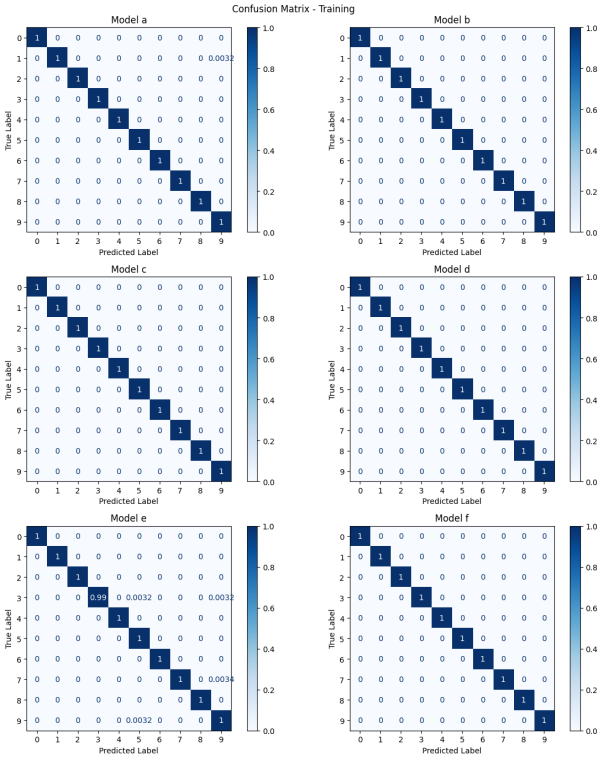

Fig. 5: Matrices de Confusión Normalizadas en conjunto de entrenamiento.

E. Matrices de Confusión - Validación

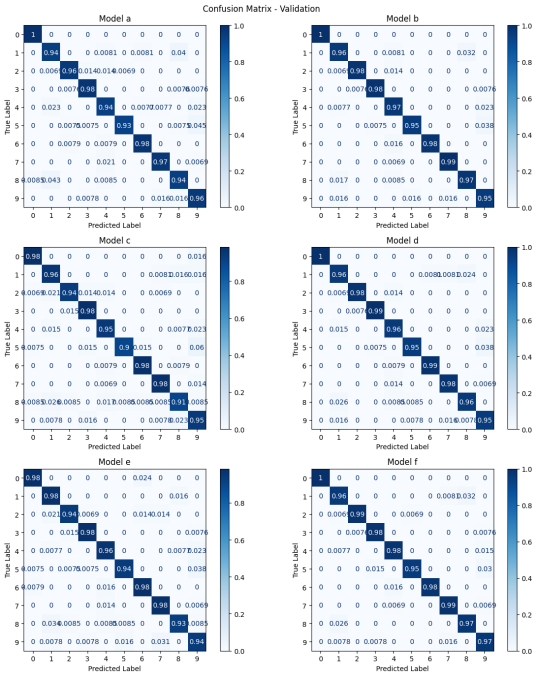

Fig. 6: Matrices de Confusión Normalizadas en conjunto de validación.

F. Mejor modelo

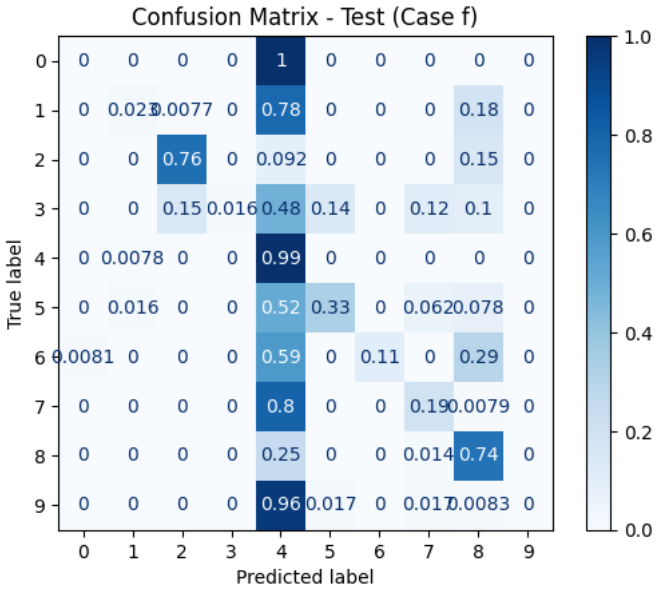

Fig. 7: Matriz de Confusión y Accuracy Modelo F en el conjunto de prueba.

V. Análisis

A. Análisis de los códigos de entrenamiento de las redes

Modelos de los cuales se puede inferir que el incremento de la cantidad de neuronas en la capa oculta afecta positivamente si se combina con la función de activación correcta, por ejemplo con la función de activación Tanh y 10 neuronas no se nota gran cambio y el tiempo de ejecución es lento pero si se aumentan las neuronas y capas ocultas, se mejora el doble obteniendo un tiempo decente. A su vez, en el modelo f como se vió anteriormente, con la función de activación ReLU, dos capas ocultas y 40 neuronas cada una, se logra el mejor entrenamiento.

B. Con respecto a los gráficos de la subsección Loss

Los gráficos de pérdida de entrenamiento y validación muestran la evolución de la pérdida del modelo a medida que se entrena y se valida. En general, se espera que la pérdida de entrenamiento disminuya a medida que el modelo aprende, mientras que la pérdida de validación debería permanecer relativamente constante o aumentar ligeramente. Esto se debe a que el modelo está aprendiendo a generalizar a datos nuevos, lo que puede aumentar la pérdida.

En el caso de los gráficos proporcionados, se observa que la pérdida de entrenamiento de todos los modelos disminuye a medida que se entrenan. Sin embargo, la pérdida de validación de los modelos A y B comienza a aumentar después de un cierto número de épocas. Esto sugiere que estos modelos están comenzando a sobreajustarse a los datos de entrenamiento.

El modelo f tiene la curva de pérdida de validación más estable, lo que sugiere que es menos propenso a sobreajustarse. Este modelo también tiene la pérdida de entrenamiento más baja, lo que sugiere que es el modelo más preciso.

C. Luego, para la Precisión

El gráfico proporcionado muestra la evolución de la precisión del modelo a medida que se entrena y se valida. La precisión de entrenamiento se define como la proporción de muestras de entrenamiento que el modelo clasifica correctamente. La precisión de validación se define como la proporción de muestras de validación que el modelo clasifica correctamente.

En general, se espera que la precisión de entrenamiento aumente a medida que el modelo aprende, mientras que la precisión de validación debería permanecer relativamente constante o aumentar ligeramente. Esto se debe a que el modelo está aprendiendo a generalizar a datos nuevos, lo que puede aumentar la precisión.

D. Análisis de los tiempos de ejecución

El gráfico muestra los tiempos de entrenamiento de los seis modelos de redes neuronales. Los modelos se etiquetan del A al F. En el eje X tenemos las etiquetas de los modelos, mientras que en el eje Y el tiempo de ejecución en segundos

El gráfico muestra que el modelo A es el más lento para entrenar, con un tiempo de entrenamiento de 35.677886 segundos. El modelo F es el más rápido para entrenar, con un tiempo de entrenamiento de 13.563620 segundos.

Se puede concluir que el modelo F es el modelo más eficiente en términos de tiempo de entrenamiento. Este modelo es ideal para aplicaciones donde el tiempo de entrenamiento es un factor importante.

E. Análisis de las Matrices de Confusión - Entrenamiento

Como bien se puede apreciar, se arroja una diagonal de 1's y ceros alrededor lo cual es muy poco probable por lo que apunta a un error de normalización o en la evaluación de la matriz de confusión, al menos en el modelo e hay una pequeña variación de datos.

F. Análisis de las Matrices de Confusión - Validación

En general, se espera que la diagonal principal de una matriz de confusión tenga los valores más altos. Esto se debe a que las predicciones correctas deben estar en la diagonal principal.

Se puede observar que el modelo F tiene la matriz de confusión más precisa. Este modelo tiene la mayoría de los valores en la diagonal principal, lo que sugiere que está haciendo predicciones precisas. Mientras que el modelo A tiene la matriz de confusión menos precisa. Este modelo tiene muchos valores fuera de la diagonal principal, lo que sugiere que está haciendo muchas predicciones incorrectas.

Ahondemos en el modelo F:

La matriz de confusión del modelo F muestra que el modelo tiene una precisión del 99%. Esto significa que el modelo clasifica correctamente el 99%de las muestras de validación.

La matriz de confusión también muestra que el modelo tiene una sensibilidad del 99% y una especificidad del 99%. La sensibilidad se define como la proporción de muestras positivas que el modelo clasifica correctamente. La especificidad se define como la proporción de muestras negativas que el modelo clasifica correctamente.

En este caso, la sensibilidad y la especificidad del modelo F son iguales, lo que indica que el modelo está clasificando correctamente tanto las muestras positivas como las negativas.

G. Matriz de Confusión mejor modelo, en el conjunto de prueba

Esta sección revela una discrepancia significativa, ya que el rendimiento del modelo al enfrentarse a datos nuevos fue notablemente pobre en comparación con sus resultados en el entrenamiento y la validación. Al observar la matriz de confusión de la Figura 7, se evidencia que en algunas clases, como la 0 y 9, el modelo no logró clasificar ningún dato de manera correcta, aún así logra valores aceptables en las coordenadas (2,2), (4.4) y (8.8). Además, el índice de precisión en el conjunto de prueba resulta notablemente bajo como se observa a continuación:

```
Accuracy - Test (Case f): 0.31761006289308175
```

Fig. 8: índice de precisión en el conjunto de prueba.

En comparación con los resultados de las fases de entrenamiento y validación. Estos resultados podrían sugerir que a pesar de implementar el early stopping, el modelo está sobreajustado y no generaliza eficientemente con datos no vistos.

VI. Conclusiones Generales

El estudio de clasificación de dígitos manuscritos mediante redes neuronales ha proporcionado resultados prometedores y ha identificado áreas clave para mejorar la eficiencia y precisión de los modelos. Se empleó PyTorch para la carga de datos y la construcción de diversas arquitecturas de redes neuronales, las cuales fueron entrenadas con un límite de 1000 épocas y se aplicó la técnica de early stopping para evitar el sobreajuste.

En cuanto a la pérdida y precisión, se observó que algunos modelos mostraron signos de sobreajuste, especialmente los modelos A y B, mientras que el modelo F demostró una trayectoria de pérdida más estable y mayor precisión.

Los tiempos de entrenamiento variaron entre modelos, siendo el modelo F el más eficiente. Esto lo posiciona como una excelente opción para aplicaciones donde la rapidez en el entrenamiento es crítica.

El análisis de las matrices de confusión mostró que el modelo F mantuvo altos niveles de precisión y exactitud tanto en el conjunto de entrenamiento como en el de validación. Sus valores altos en la diagonal principal indican una clasificación precisa y confiable.

Sin embargo, al evaluar el conjunto de prueba, se evidenció un rendimiento inferior en comparación con las fases de entrenamiento y validación. Los resultados sugieren una limitación en la capacidad de generalización de los modelos a datos no vistos.

Este análisis resalta la importancia de evitar el sobreajuste y mejorar la capacidad de generalización de los modelos de redes neuronales para obtener predicciones precisas y consistentes en entornos del mundo real. Se subraya la relevancia de modelos eficientes y con buena capacidad de generalización, como el modelo F, para la construcción de aplicaciones de aprendizaje automático confiables.

VII. Bibliografía

https://aws.amazon.com/es/what-is/neural-network/

https://www.diegocalvo.es/funcion-de-activacion-redes-neuronales/#:~:text=ReLU%20%20Rectified%20Lineal%20Unit,positivos%20tal%20y%20como%20entran.&text=Caractersticas%20de%20la%20funcin%20ReLU,No%20est%20acotada.

https://www.codificandobits.com/blog/matriz-de-confusion/

https://pytorch.org/docs/stable/nn.html#linear-layers