Universidad de O'Higgins

Tarea N° 2:
Redes neuronales

José Tomás Garciá Risco

COM4402 – Introducción a Inteligencia Artificial

Escuela de Ingeniería, Universidad de O'Higgins

09, Octubre, 2023

***Abstract*—Aquí se espera mostrar diferentes de modelos de redes neuronales y su desempeño según diferentes funciones de activación además de topología especificamente Tanh y ReLU dentro de una base de datos creada utilizando programas de preprocesamiento proporcionados por el Instituto Nacional de Estándares y Tecnología (NIST) para extraer imágenes normalizadas de dígitos escritos a mano a partir de un formulario preimpreso.. Se usará Python como lenguaje de programación junto con las librerías Sklearn, Numpy Pandas y Torch para el análisis de datos.**

Palabras Clave: Redes neuronales,Python ,Overfitting ,Épocas, Loss

I. Introduction

La clasificación de dígitos escritos a mano es un desafío clásico en el campo de la visión por computadora y el aprendizaje automático. La capacidad de reconocer y diferenciar dígitos manuscritos es fundamental en una amplia variedad de aplicaciones, desde el procesamiento de cheques y formularios hasta la identificación de caracteres en sistemas de lectura óptica de caracteres (OCR).

El conjunto de datos "Optical Recognition of Handwritten Digits", que contiene imágenes normalizadas de dígitos escritos a mano, es una herramienta valiosa para abordar este desafío. Sin embargo, para aprovechar al máximo este conjunto de datos, es necesario emplear técnicas de aprendizaje automático avanzadas, y aquí es donde entran en juego las redes neuronales.

Las redes neuronales artificiales, inspiradas en el funcionamiento del cerebro humano, son modelos computacionales capaces de aprender y extraer patrones complejos a partir de datos. En este informe, explicaremos cómo las redes neuronales se aplican al problema de clasificación de dígitos utilizando el conjunto de datos mencionado. Utilizaremos la estructura de las redes neuronales para extraer automáticamente características relevantes de las imágenes y tomar decisiones de clasificación precisas.

A medida que avancemos en este informe, analizaremos en detalle la metodología empleada para entrenar y evaluar estas redes neuronales, así como los resultados obtenidos en términos de precisión y capacidad de generalización. Además, discutiremos la importancia de la elección de la función de activación, la profundidad de la red y otros factores críticos en el rendimiento de nuestro modelo.

II. Marco Teorico

- Red neuronal: Las redes neuronales son un modelo computacional evolucionado a partir de diversas contribuciones científicas. Consiste en un conjunto de unidades llamadas neuronas artificiales, conectadas entre sí para transmitir señales.
- Capas de Neuronas: Una red neuronal suele estar compuesta por múltiples capas, incluyendo una capa de entrada, una o varias capas ocultas y una capa de salida. Cada neurona en una capa está conectada a todas las neuronas en la capa anterior y posterior.
- Funciones de Activación: Cada neurona aplica una función de activación a su entrada ponderada. Estas funciones introducen no linealidad en la red, lo que le permite aprender relaciones no lineales en los datos.
- Entrenamiento: Las redes neuronales se entrenan utilizando algoritmos de

optimización que ajustan los pesos y sesgos para minimizar una función de pérdida. El aprendizaje se basa en la retropropagación del error, donde los errores se propagan hacia atrás a través de la red para ajustar los parámetros.

- Neurona Artificial:Una neurona artificial es la unidad básica de una red neuronal. Recibe entradas, aplica una función de activación a la suma ponderada de esas entradas y produce una salida.
- Capas Ocultas:En una red neuronal, las capas ocultas son capas intermedias entre la capa de entrada y la capa de salida. Estas capas son responsables de aprender representaciones intermedias de los datos, lo que permite a la red modelar relaciones más complejas.
- Capa de Salida:La capa de salida de una red neuronal es la última capa y produce la salida final del modelo. En problemas de clasificación, esta capa a menudo tiene una neurona por clase y produce probabilidades de pertenencia a cada clase.
- Loss (Pérdida):La función de pérdida (loss) cuantifica la diferencia entre las predicciones del modelo y los valores reales en el conjunto de datos. El objetivo es minimizar esta pérdida durante el entrenamiento.
- Optimizador:El optimizador es un algoritmo que ajusta los parámetros de la red neuronal para minimizar la pérdida. En este caso usamos Adam.
- Conjuntos de Entrenamiento, Validación y Prueba: Los conjuntos de entrenamiento se utilizan para entrenar el modelo. Los conjuntos de validación se utilizan para ajustar hiper parámetros y prevenir el sobreajuste. Los conjuntos de prueba se utilizan para evaluar el rendimiento del modelo en datos no vistos.
- DataLoaders y Batches:Los DataLoaders son utilidades que dividen el conjunto de datos en lotes (batches) más pequeños para el entrenamiento. Esto facilita el procesamiento eficiente de grandes conjuntos de datos y mejora la generalización.
- Overfitting (Sobreajuste):El sobreajuste ocurre cuando un modelo se ajusta demasiado a los datos de entrenamiento y tiene dificultades para generalizar a nuevos datos. Puede ocurrir cuando el modelo es demasiado complejo o cuando hay pocos datos de entrenamiento.
- ReLU (Rectified Linear Unit): ReLU es una función de activación no lineal que se utiliza comúnmente en redes neuronales. Introduce no linealidad al ser cero para valores negativos y lineal para valores positivos. Ayuda a la red a aprender representaciones no lineales.
- Tanh (Tangente Hiperbólica):Tanh es otra función de activación no lineal que mapea los valores de entrada a un rango entre -1 y 1. Es útil en redes neuronales para introducir no linealidad y preservar la dirección de los datos.
- Accuracy (Precisión):La precisión es una métrica de evaluación que mide la proporción de predicciones correctas realizadas por el modelo en el conjunto de prueba. Se expresa como el número de predicciones correctas dividido por el número total de predicciones.

III. METODOLOGÍA

Empezamos importando las librerías y luego descargamos los datos de un repositorio proporcionado , una vez con ellos empezamos el preprocesamiento de datos, los separamos en conjuntos de entrenamiento,prueba y validación estandarizamos las características que se utilizan, para asegurarse de que estas tengan una escala consistente.

```
column_names = ["feat" + str(i) for i in range(64)]
column_names.append("class")
```

```
df_train_val = pd.read_csv('1_digits_train.txt', names = column_names
df_train_val
```

```
df_test = pd.read_csv('1_digits_test.txt', names = column_names)
df_test
```

```
df_train, df_val = train_test_split(df_train_val, test_size = 0.3, ra
```

```
scaler = StandardScaler().fit(df_train.iloc[:,0:64])
df_train.iloc[:,0:64] = scaler.transform(df_train.iloc[:,0:64])
df_val.iloc[:,0:64] = scaler.transform(df_val.iloc[:,0:64])
df_test.iloc[:,0:64] = scaler.transform(df_test.iloc[:,0:64])
```

Luego definimos el modelo ,el dispositivo en el que se realizará el entrenamiento de la red neuronal. En este caso, se utiliza 'cuda', que se refiere a una GPU, la función de pérdida que se utilizará para medir la diferencia entre las predicciones del modelo CrossEntropyLoss es una función de pérdida comúnmente utilizada en problemas de clasificación y es adecu y se crea un optimizador que se utilizará para ajustar los parámetros de la red neuronal durante el proceso de entrenamiento. En este caso, se utiliza el optimizador Adam . El optimizador Adam es un algoritmo de optimización que se utiliza comúnmente para entrenar redes neuronales debido a su eficacia y velocidad de convergencia

```
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)
```

creamos los dataset y dataloaders para nuestros conjuntos estos nos permitirán evaluar la efectividad del modelo simplifican la tarea de dividir los datos en lotes (batches) y proporcionan una forma conveniente de cargar los datos en el modelo de manera eficiente

```
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":labe

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_

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
```

```
# Crear dataloaders
dataloader_train = torch.utils.data.DataLoader(dataset_train,
dataloader_val = torch.utils.data.DataLoader(dataset_val, bat
dataloader_test = torch.utils.data.DataLoader(dataset_test, b
```

Empezamos el entrenamiento del modelo e iniciamos un temporizador para realizar un seguimiento del tiempo de entrenamiento. También se inicializan variables para el mejor accuracy en validación, listas para almacenar pérdidas y precisión en entrenamiento y validación, y se definen parámetros para el criterio de parada temprana,

```
start = time.time()

best_val_accuracy = 0
loss_train = []
loss_val = []
accuracy_train = []  # Para almace
accuracy_val = []    # Para almace
epochs = []

early_stopping_patience = 10 #nume
min_loss_val = float('inf')
patience = 0
```

El entrenamiento se realiza durante un número máximo de épocas (en este caso, hasta 1000 épocas). Un epoch es una pasada completa a través de todo el conjunto de entrenamiento.Dentro de cada época, se itera a través de los lotes (batches) del conjunto de

entrenamiento utilizando un DataLoader. En cada lote, se realizan las siguientes acciones:

- Se envían las características y etiquetas al dispositivo (por ejemplo, GPU) si se ha configurado previamente.
- Se calculan las predicciones del modelo, la pérdida y se realiza la retropropagación (backpropagation) para ajustar los parámetros del modelo.
- Se almacena la pérdida y se calcula la precisión (accuracy) en entrenamiento para el lote actual.

```
for epoch in range(1000):
  loss_train_batches = []
  loss_val_batches = []
  accuracy_train_batches = []  # Para almacenar el accuracy
  accuracy_val_batches = []    # Para almacenar el accuracy
  # Entrenamiento
  model.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) # Características
    labels = data["labels"].to(device)   # Clases
    # zero the parameter gradients
    optimizer.zero_grad()
    # forward + backward + optimize
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward() # backpropagation
    optimizer.step()

    # Por completar: calcule la pérdida de validación y acu
    loss_train_batches.append(loss.item())
    accuracy_batch = accuracy_score(labels.cpu().numpy(), c
    accuracy_train_batches.append(accuracy_batch)  # Almace
```

Se cambia el modelo a modo de evaluación (.eval()) y se evalúa en el conjunto de validación.

```
model.eval()
true_labels_train = []
predicted_labels_train = []
with torch.no_grad():
  # Iteramos dataloader_val para evaluar el modelo en los datos d
  for i, data in enumerate(dataloader_val, 0):
    # Procesar batch actual
    inputs = data["features"].to(device) # Características
    labels = data["labels"].to(device)   # Clases
    outputs = model(inputs)              # Obtenemos predicciones
    # Guardamos la pérdida de validación en el batch actual
    loss = criterion(outputs, labels)
    loss_val_batches.append(loss.item())
    accuracy_batch = accuracy_score(labels.cpu().numpy(), outputs
    accuracy_val_batches.append(accuracy_batch)  # Almacena la pr
```

Si el accuracy en validación en la época actual es mejor que el mejor accuracy en validación anterior, se guarda el modelo en un archivo llamado "best_model.pth" ademas se verifica si la pérdida en validación ha dejado de disminuir. Si no hay mejora durante un número específico de épocas (definido por early_stopping_patience), el entrenamiento se detiene antes de alcanzar el límite de épocas.

```
if epoch_val_accuracy > best_val_accuracy:
    best_val_accuracy = epoch_val_accuracy
    torch.save(model.state_dict(), 'best_model.pth')
if loss_val_epoch < min_loss_val:
    min_loss_val = loss_val_epoch
    patience = 0  # Restablecer el contador de pacie
else:
    patience += 1
```

y finaliza el entrenamiento del modelo y creamos los gráficos correspondientes para el análisis de loss y accuracy del modelo

IV. RESULTADOS

- 10 neuronas en la capa oculta, usando función de activación ReLU y 1000 épocas como máximo: tiende a necesitar más de 100 epochs baja loss muy buena precisión en los 3 modelos tambien el mas demoroso
- Modelo 40 neuronas en la capa oculta y función de activación ReLU, y 1000 épocas como máximo: necesita menos epochs que el el de 10 neuronas además de tener mejores resultados en precisión y necesitar menos tiempo de entrenamiento
- Modelo 10 neuronas en la capa oculta y función de activación Tanh, y 1000 épocas como máximo: a diferencia de ReLU tiende a necesitar pasar por más epochs por lo que también necesita un mayor tiempo de entrenamiento pero tiende a mostrar una mayor precisión a cambio
- Modelo 40 neuronas en la capa oculta y función de activación Tanh, y 1000 épocas como máximo: presenta resultados muy parecidos al modelo de ReLU de forma general

- Modelo 40 neuronas en la capa oculta y función de activación Tanh, y 1000 épocas como máximo: el modelo donde el loss baja de la forma mas rapida ademas de presentar la mejor precisión para el label 8 y 9 que tienden a ser donde los demás modelos tienen más problemas fuera de eso tiene gran parecido a los otros modelos de 40 neuronas

V. ANÁLISIS

- ReLU: tiende a ser el más rápido de los modelos pero a veces sacrifica un poco de precisión por esto.
- Tanh: se le podría llamar más consistente al poder pasar por más epochs antes del early stopping además de tender a presentar una mayor precisión
- 10 vs 40 neuronas: los modelos de 40 neuronas presentaron mejor precisión y menor pérdida de validación esto puede ser por que al tener más neuronas en la capa oculta tienen una mayor capacidad de representación además de tener una generalización mejorada sin el riesgo de sobreajuste gracias al early stop

En general los datos proporcionados parecen prestarse a trabajar con redes neuronales ya sea por patrones reconocibles o significativos estos se ajustan bien a todos los modelos usados

VI. CONCLUSIONES

Pudimos encontrar que ciertos modelos se ajustan mejor que otros para ciertos datos y como un mayor número de neuronas en diferentes capas puede beneficiar ciertos parámetros más que otros.