**PRACTICA #1 Clasificación Básica: Predecir una imagen de moda**

La clasificación de imágenes es una tarea fundamental en el ámbito de la visión por computadora. En esta práctica, exploraremos el uso de una red neuronal totalmente conectada para abordar el desafío de clasificar el popular conjunto de datos *Fashion MNIST*. Este conjunto de datos consta de imágenes en escala de grises de prendas de vestir en 10 categorías diferentes, y el objetivo es desarrollar un modelo capaz de asignar automáticamente estas imágenes a sus respectivas clases.

Para lograr esto, implementaremos una red neuronal bastante sencilla , la cual se basa en un enfoque de aprendizaje profundo mediante capas de neuronas completamente conectadas. Esta red consta de una capa de entrada que acomoda las imágenes después de ser aplanadas, seguida de una capa oculta con función de activación ReLU para capturar características relevantes, y finalmente, una capa de salida que produce distribuciones de probabilidad sobre las clases de prendas de vestir.

**Código en Tensorflow:**

model = keras.Sequential([

keras.layers.Flatten(input\_shape=(28, 28)),

keras.layers.Dense(128, activation='relu'),

keras.layers.Dense(10, activation='softmax')

])

**Codigo en Pytorch:**

class SimpleNN(nn.Module):

def \_\_init\_\_(self):

super(SimpleNN, self).\_\_init\_\_()

self.flatten = nn.Flatten()

self.linear1 = nn.Linear(28 \* 28, 128)

self.relu = nn.ReLU()

self.linear2 = nn.Linear(128, 10)

def forward(self, x):

x = self.flatten(x)

x = self.linear1(x)

x = self.relu(x)

x = self.linear2(x)

return x

La arquitectura comprende tres componentes clave:

1. **Aplanamiento de Datos:** Dado que las imágenes son matrices 2D, la primera capa realiza el aplanamiento de cada imagen en un vector unidimensional, permitiendo que la red neuronal procese los píxeles como características individuales.
2. **Capa Oculta:** La segunda capa, compuesta por 128 neuronas, se encarga de aprender representaciones abstractas de las características presentes en las imágenes a través de la función de activación ReLU. Esta capa intermedia es esencial para capturar relaciones no lineales en los datos.
3. **Capa de Salida:** La última capa consta de 10 neuronas, cada una representando una clase de prenda de vestir distinta. La función de activación utilizada aquí es lineal para pytorch, Y en el caso de Tensorflow la funcion de activacion es softmax directamente. En muchos casos se recomienda utilizar una capa de salida lineal y combinarla con la función de pérdida en caso pytorch “nn.CrossEntropyLoss” en lugar de aplicar directamente una función softmax1. Esto se debe a varias razones:
   1. **Estabilidad Numérica:** Utilizar una capa de salida lineal sin la función softmax previa puede mejorar la estabilidad numérica durante el cálculo de los gradientes y los valores de salida. La función softmax puede a menudo generar valores muy grandes que pueden llevar a problemas numéricos.
   2. **Función de Pérdida CrossEntropyLoss:** La función de pérdida nn.CrossEntropyLoss combina la aplicación de la función softmax y el cálculo de la entropía cruzada en una sola etapa. Por lo tanto, usar una capa de salida lineal en combinación con esta función de pérdida es más eficiente y puede llevar a un entrenamiento más estable.

**Selección del Optimizador y Función de Pérdida**

Después de implementar la arquitectura de la red neuronal y definir la estructura de las capas, es esencial elegir apropiadamente el optimizador y la función de pérdida para entrenar y evaluar el modelo. En este caso, se seleccionó el optimizador Adam y la función de pérdida sparse\_categorical\_crossentropy/CrossEntropyLoss para llevar a cabo el entrenamiento de la red neuronal .

**Optimizador Adam**

El optimizador Adam (Adaptive Moment Estimation) es una elección popular para entrenar redes neuronales debido a su eficiencia y efectividad en una variedad de problemas. Adam combina las ventajas de los optimizadores de primer orden como el descenso de gradiente estocástico (SGD) con mecanismos de ajuste adaptativo de la tasa de aprendizaje. Esto significa que el optimizador es capaz de adaptarse automáticamente a diferentes tasas de aprendizaje para cada parámetro del modelo3.

**Función de Pérdida**

La función de pérdida sparse\_categorical\_crossentropy/CrossEntropyLoss es la elección apropiada para problemas de clasificación multiclase, como en el caso de Fashion MNIST, donde cada imagen pertenece a una de las diez categorías de prendas de vestir.

**Codigo en Tensorflow:**

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

**Codigo en Pytorch:**

lr = 1e-3

model = SimpleNN()

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=lr)

**Preparación y Preprocesamiento del Conjunto de Datos**

Antes de alimentar cualquier conjunto de datos a un modelo de aprendizaje automático, es crucial realizar un proceso de preparación y preprocesamiento para garantizar que los datos sean adecuados y estén en un formato que el modelo pueda entender y aprender.

**Balance de Clases**

Una característica importante del conjunto de datos es que está balanceado, lo que significa que cada una de las 10 clases de prendas de vestir tiene la misma cantidad de ejemplos. En el conjunto de entrenamiento, hay 6,000 ejemplos de cada clase, y en el conjunto de pruebas, hay 1,000 ejemplos por clase. Este equilibrio es esencial para evitar sesgos en el entrenamiento y asegurar que el modelo aprenda características útiles de todas las clases por igual.

**Normalización de Datos**

Una práctica común en el preprocesamiento de imágenes es normalizar los datos antes de alimentarlos al modelo. En este caso, los valores de píxeles de las imágenes se normalizan dividiendo cada valor por 255. Esto tiene una importancia significativa debido a la escala de los valores de píxeles en una imagen en escala de grises. Al normalizar los valores en el rango de [0, 1], se logra lo siguiente:

1. **Mejor Convergencia:** La normalización ayuda a que el proceso de entrenamiento sea más estable y converge más rápido. Las diferencias en la escala de los datos pueden ralentizar el proceso de optimización.
2. **Reducción del Impacto de los Valores Atípicos:** Los valores atípicos o ruidosos tienen menos impacto en el entrenamiento después de la normalización, ya que todos los valores están en el mismo rango.
3. **Preparación para Funciones de Activación:** Muchas funciones de activación, como la función ReLU, funcionan mejor cuando los datos están en un rango acotado.

**Codigo en Tensorflow:**

train\_images.shape

test\_images.shape

train\_images = train\_images / 255.0

test\_images = test\_images / 255.0

**Codigo en Pytorch:**

print(x\_train.shape, y\_train.shape)

print(x\_test.shape, y\_test.shape)

for i in range(0,10):

print(f'la etiqueta {i} tiene {y\_train[ y\_train == i].shape[0]} datos')

x\_train = torch.tensor(x\_train, dtype = torch.float32) / 255.0

x\_test = torch.tensor(x\_test, dtype = torch.float32) / 255.0

**Técnica de Mini Lotes (Mini-Batch):**

La técnica de mini lotes implica dividir el conjunto de datos completo en pequeños lotes más manejables para el procesamiento. Cada lote contiene un número reducido de ejemplos. En lugar de alimentar todos los datos de entrenamiento a la red neuronal de una sola vez (técnica conocida como entrenamiento de lote completo o batch gradient descent), se actualizan los pesos del modelo después de cada mini lote 2.

**Importancia de los Mini Lotes:**

1. **Eficiencia en Memoria:** Algunos conjuntos de datos pueden ser demasiado grandes para caber completamente en la memoria. Usar mini lotes permite entrenar modelos incluso con recursos limitados, ya que solo se necesita cargar un pequeño lote de datos a la vez.
2. **Optimización Estocástica:** El uso de mini lotes introduce un elemento de aleatoriedad en el proceso de optimización. En cada paso, el modelo se actualiza con un subconjunto de datos aleatorio, lo que puede ayudar a evitar que el modelo se atasque en óptimos locales.
3. **Paralelización:** Los mini lotes también permiten aprovechar la capacidad de procesamiento paralelo de las unidades de procesamiento gráfico (GPU) y las unidades de procesamiento central (CPU), lo que acelera el entrenamiento.
4. **Mayor Estabilidad:** La variabilidad introducida por los mini lotes puede ayudar a que el proceso de entrenamiento sea más estable, ya que cada lote puede proporcionar información útil sobre la dirección del gradiente.

**Codigo en Tensorflow:**

No se aplico

**Codigo en Pytorch:**

train\_dataset = TensorDataset(x\_train, y\_train)

batch\_size = 64

train\_dataloader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

**Entrenando el Modelo de Clasificación con Fashion MNIST**

Después de haber preparado y preprocesado el conjunto de datos Fashion MNIST, y de haber definido el modelo de red neuronal, estamos listos para entrenar el modelo para realizar la clasificación de prendas de vestir. El proceso de entrenamiento se lleva a cabo en varias etapas, donde el modelo aprende a ajustar sus pesos y sesgos de manera que pueda hacer predicciones precisas sobre las imágenes de prueba.

**Codigo en Tensorflow:**

model.fit(train\_images, train\_labels, epochs=10)

**Codigo en Pytorch:**

nb\_epochs = 10

training\_loss = training(model, optimizer, criterion,

nb\_epochs, train\_dataloader)

def training(model, optimizer, criterion,

nb\_epochs, data\_loader):

training\_loss = []

for epoch in range(nb\_epochs):

for batch\_images, batch\_labels in data\_loader:

optimizer.zero\_grad()

outputs = model(batch\_images)

loss = criterion(outputs, batch\_labels)

loss.backward()

optimizer.step()

training\_loss.append(loss.item())

print(f"Epoch [{epoch+1}/{nb\_epochs}], Loss: {loss.item():.4f}")

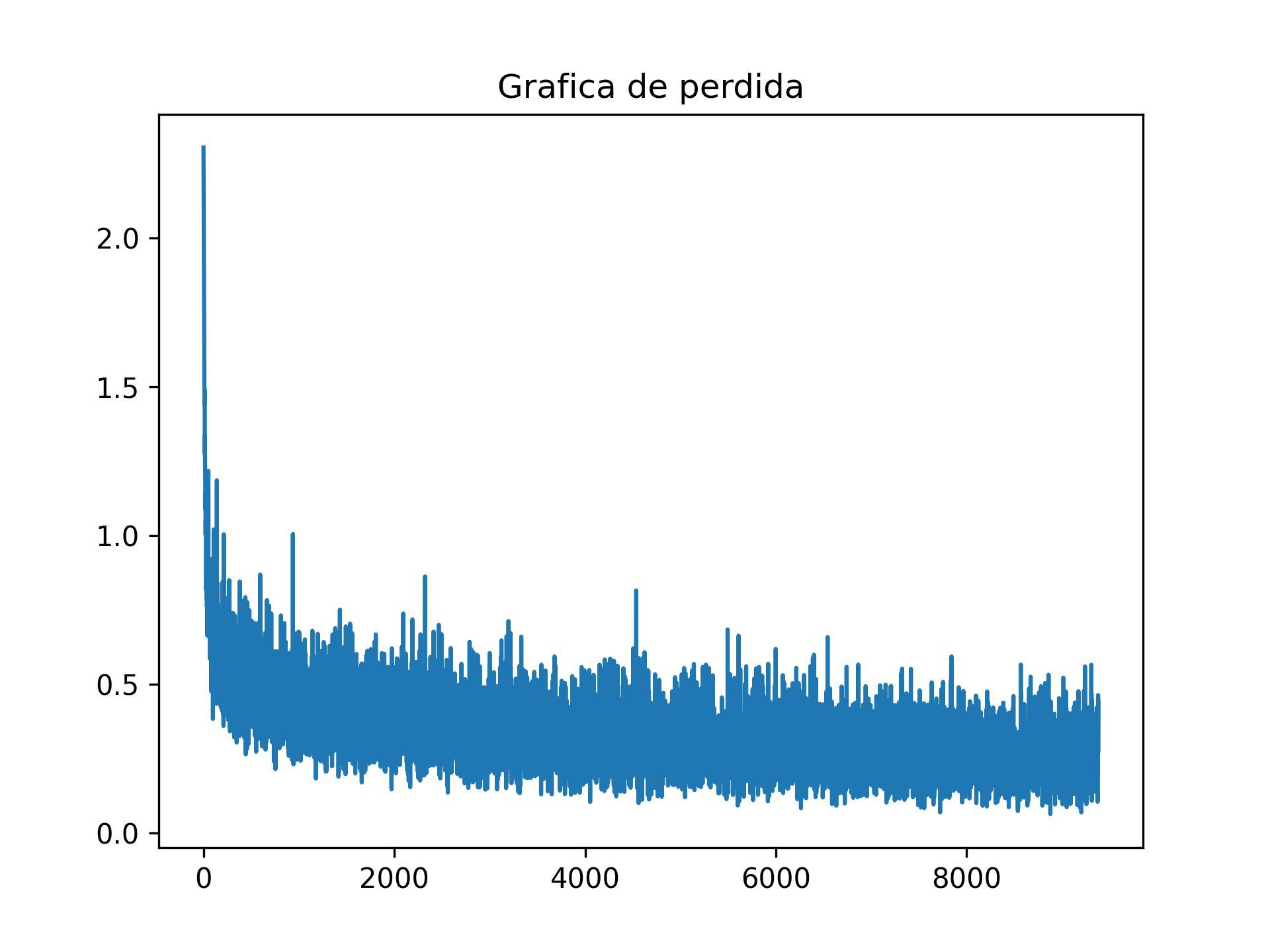
return training\_loss

**Resultados: Evaluación del Modelo Entrenado**

Después de completar el proceso de entrenamiento de nuestro modelo de clasificación utilizando Fashion MNIST, es fundamental evaluar su desempeño y comprender cómo se ha generalizado a datos no vistos.

**Gráfica de Pérdida:**

La gráfica de pérdida es una herramienta visual que nos proporciona información sobre cómo la pérdida (error) del modelo cambia a lo largo de las épocas de entrenamiento. Si todo va bien, esperamos ver una disminución gradual en la pérdida a medida que el modelo ajusta sus parámetros para hacer predicciones más precisas.



**Evaluación de la Exactitud en los Datos de Prueba:**

La pérdida es una medida útil para comprender cómo el modelo está mejorando internamente, pero la métrica más directa para evaluar el desempeño de un modelo de clasificación es la exactitud. Esta métrica nos indica la proporción de predicciones correctas en relación con el total de predicciones realizadas en el conjunto de datos de prueba.

**Codigo en Tensorflow:**

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels, verbose=2)

print('\nTest accuracy:', test\_acc)

**Codigo en Pytorch:**

test\_dataset = TensorDataset(x\_test, y\_test)

batch\_size = 64

test\_dataloader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)

error\_idx, predicted = test(model, test\_dataloader)

def test(model, data\_loader):

correct = 0

total = 0

error\_idx = []

predicted\_list = []

with torch.no\_grad():

for batch\_images, batch\_labels in data\_loader:

outputs = model(batch\_images)

\_, predicted = torch.max(outputs, dim=1)

total += batch\_labels.shape[0]

correct += int((predicted == batch\_labels).sum())

error\_idx.append(predicted == batch\_labels)

predicted\_list.append(predicted)

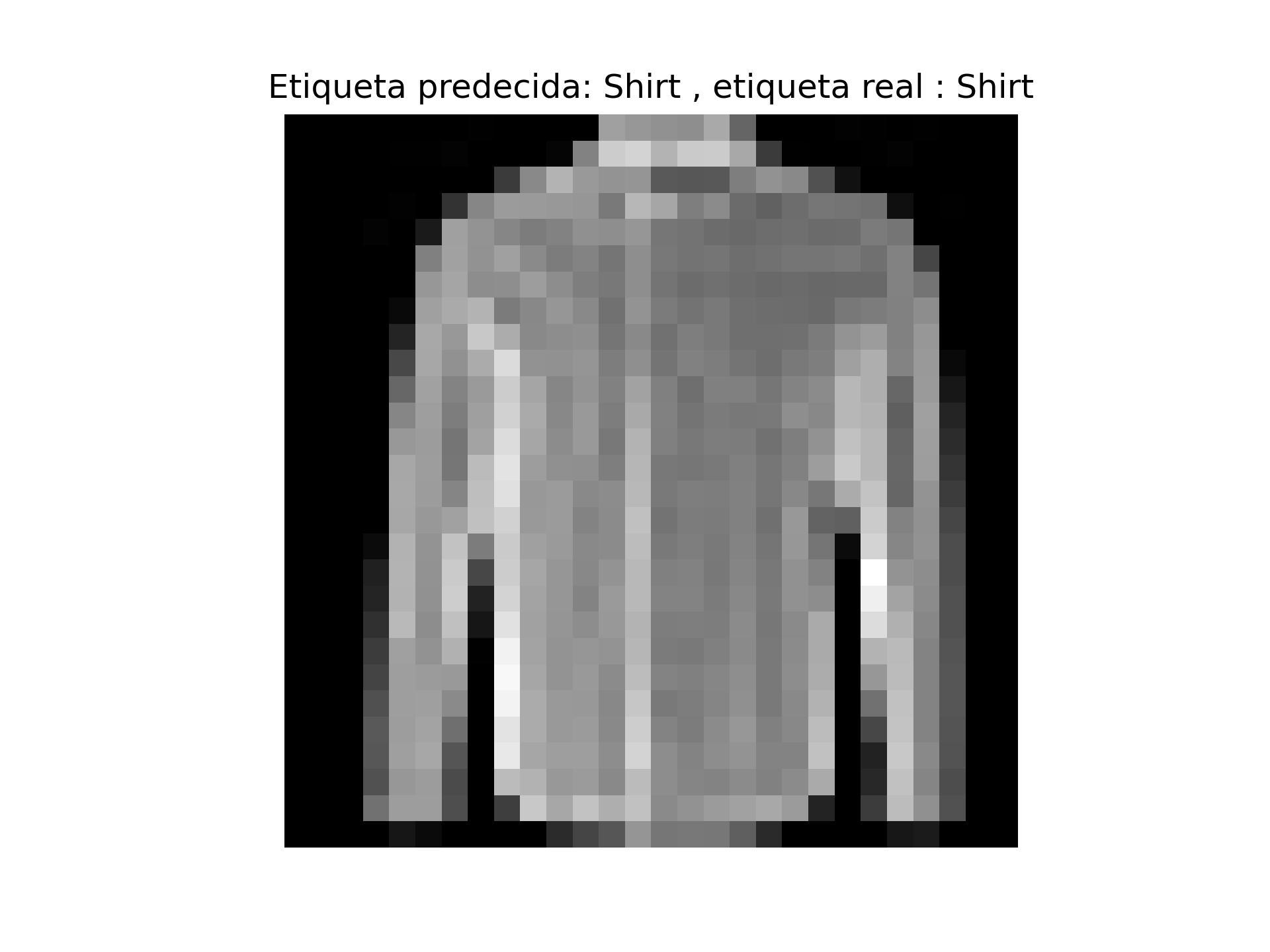
print(f"Test Accuracy: {(correct / total)\*100 }%")

return error\_idx, predicted\_list

En tensorflow se obtuvo un Test accuracy: **0.8819000124931335**.

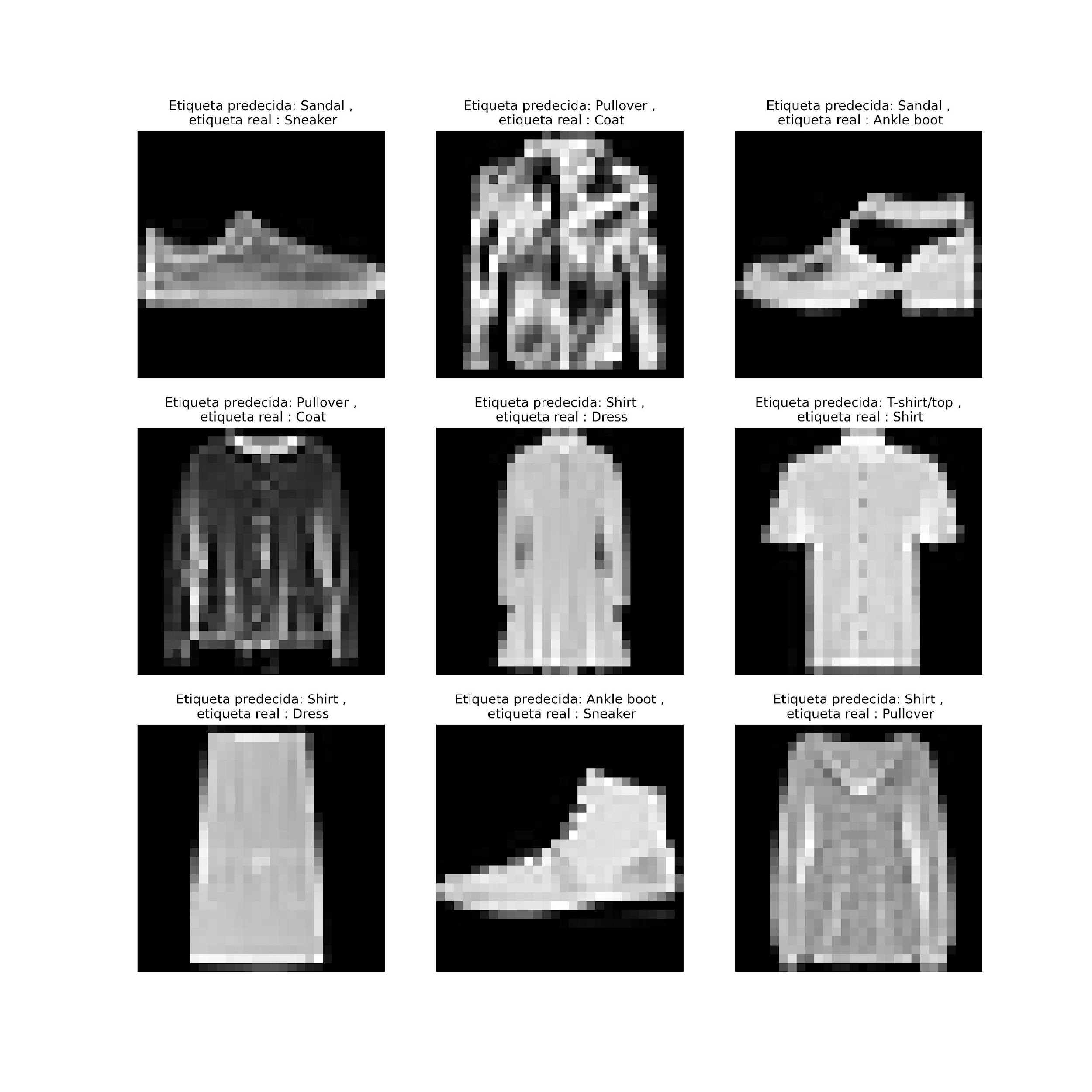
Pytorch obtuvo un Test Accuracy: **0.882** Prácticamente lo mismo.

**Prediccion en una imagen:**



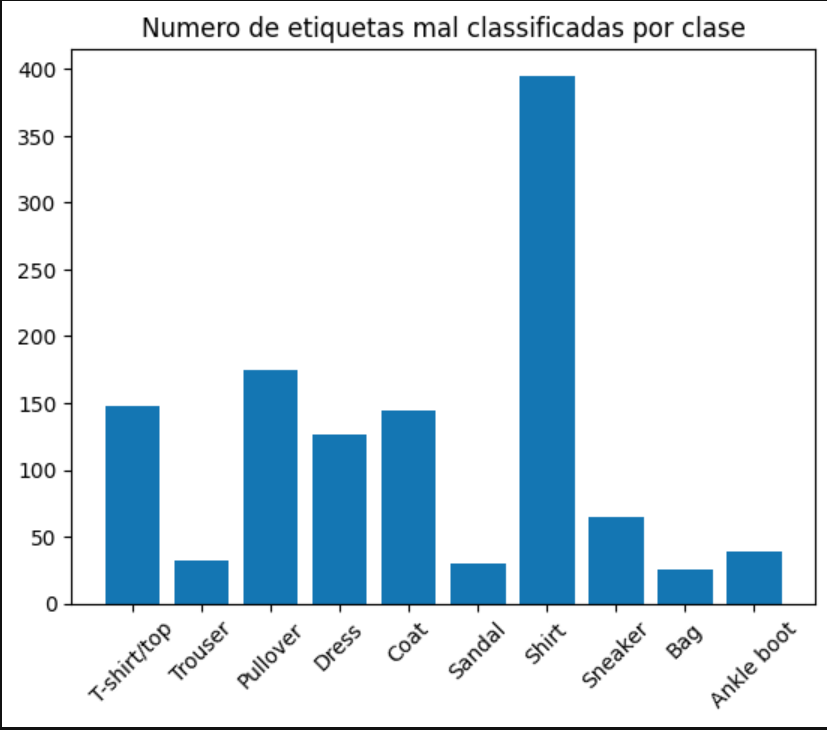
**Análisis de Predicciones Erróneas**

Durante la evaluación de nuestro modelo entrenado en el conjunto de datos de prueba, hemos identificado que se han producido **1180** predicciones incorrectas. Este resultado merece una atención especial, ya que el análisis de predicciones erróneas puede proporcionar información valiosa sobre las deficiencias del modelo y áreas en las que puede requerir mejoras.



**La cantidad de errores por clase son la siguiente:**

T-shirt/top: 148, Trouser: 32, Pullover: 175, Dress: 126, Coat: 144, Sandal: 30, Shirt: 395, Sneaker: 65, Bag: 26, Ankle boot: 39



**Conclusion:** Los resultados obtenidos en esta práctica son satisfactorios y coherentes con las expectativas establecidas. La presencia de ciertos errores en las predicciones es comprensible, dado que incluso para un observador humano resultaría un desafío distinguir con total precisión las prendas de vestir en imágenes de baja resolución de 28x28 píxeles y en escala de grises. Es relevante considerar que la red neuronal implementada tiene una arquitectura básica, y aún así lograr un nivel de precisión del 88% es un logro significativo.

Los resultados evidencian el poder del aprendizaje automático y las redes neuronales en la tarea de clasificación de imágenes, incluso con las limitaciones inherentes al conjunto de datos y la simplicidad de la arquitectura utilizada.

**Bibliografía**

1:” Advanced Learning Algorithms: Improved implementation of softmax”, by Andrew Ng. (2022), <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>

2: "ImageNet Classification with Deep Convolutional Neural Networks" by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. (2012)

3: "Adam: A Method for Stochastic Optimization" by Diederik P. Kingma and Jimmy Ba.(2015)