Proyecto 1 - Machine Learning

Este proyecto implementa un modelo de **perceptrón binario** siguiendo las especificaciones del curso de Machine Learning de la Universidad de California, Berkeley. La estructura del proyecto está contenida dentro del directorio:

- `proyectos/proyecto1`

Para ejecutar los programas, se recomienda **clonar el repositorio**. Se deben instalar las dependencias necesarias ejecutando el siguiente comando:

- `pip install -r requirements.txt`

Una vez instalado todo, se ejecutar el archivo del `autograder` con los siguientes comandos para ejecutar cada problema que se menciona en el informe:

- ```
  python autograder.py -q q1
  python autograder.py -q q2
  python autograder.py -q q3
  ```

Este desarrollo fue realizado conforme a las indicaciones del proyecto de Berkeley CS188 disponible en el siguiente enlace:

- **Berkeley CS188 - Project 5: Machine Learning**

Problema 1 - Perceptrón Binario

El **perceptrón** es uno de los modelos más simples y fundamentales del aprendizaje automático supervisado. Fue diseñado para resolver problemas de clasificación binaria, asignando a cada entrada una de dos clases posibles: `+1` o `-1`. Su funcionamiento se basa en una combinación lineal de entradas y una función de activación que determina la clase de salida.

Contexto del Problema

En este proyecto, se implementó un perceptrón binario utilizando **PyTorch**, una biblioteca eficiente para manipulación de tensores y aprendizaje automático. El objetivo del modelo es separar datos en dos clases, utilizando un vector de pesos que se ajusta iterativamente durante el entrenamiento. Para cada dato de entrada, el perceptrón calcula un puntaje y, en función de este, predice la clase correspondiente.

El modelo opera con tensores, ya que estos permiten cálculos rápidos y eficientes en múltiples dimensiones, lo cual es esencial en contextos de aprendizaje automático. Además, PyTorch ofrece funciones optimizadas como `torch.tensordot`, usada
  ```

para calcular el **producto escalar** entre el vector de pesos y el dato de entrada, que es la base para decidir la clase.

Funcionamiento del Perceptrón

El perceptrón recibe un conjunto de entradas $x_1, x_2, ..., x_n$, cada una asociada a un peso $w_1, w_2, ..., w_n$. La salida se calcula con la fórmula:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Donde b es un sesgo opcional y f es la función de activación. En el caso clásico, se utiliza una **función escalón** (también llamada función signo), que devuelve `+1` si el puntaje es mayor o igual a cero y `-1` en caso contrario.

El **puntaje** representa qué tan alineado está el dato con el vector de pesos. Su signo indica de qué lado de la frontera de decisión se encuentra el dato, y por tanto, la clase asignada.

En esta implementación, no se incluye el sesgo explícitamente, lo cual implica que la frontera de decisión pasa por el origen. Esta simplificación, aunque válida, puede limitar la capacidad del modelo para ciertos conjuntos de datos.

Entrenamiento del Perceptrón

El entrenamiento sigue una regla de aprendizaje supervisado y consiste en ajustar los pesos para minimizar los errores de clasificación. Se realiza de la siguiente manera:

1. **Inicialización de pesos**: Los pesos inician con valor `1` por requisito del autograder y también para facilitar la observación de los ajustes a lo largo del entrenamiento.
2. **Predicción**: Se calcula el puntaje usando `torch.tensordot`.
3. **Clasificación**: Se convierte el puntaje a una clase usando `torch.where(score >= 0, 1, -1)`.
4. **Corrección**: Si el dato es mal clasificado, los pesos se ajustan con la regla:

   $$\mathbf{w} \leftarrow \mathbf{w} + y \cdot \mathbf{x}$$

   Esto significa que se está moviendo la frontera de decisión para corregir el error en ese ejemplo.
5. **Repetición**: Se repite este proceso sobre todo el conjunto (una **época**), y se

continúa iterando hasta que **todos los datos sean clasificados correctamente** (convergencia) o se alcance un número máximo de iteraciones. Esto significa que el modelo sigue ajustando sus pesos hasta que logra clasificar todos los ejemplos del entrenamiento correctamente, lo que implica que el conjunto es **linealmente separable**. Si los datos no pueden separarse con una línea recta (o hiperplano en dimensiones mayores), el modelo nunca convergerá.

Nota: Durante el entrenamiento, se utiliza `with torch.no_grad()` para desactivar el cálculo de gradientes, ya que el perceptrón no requiere retropropagación.

Representación Geométrica

Geométricamente, el perceptrón busca encontrar una **recta (en 2D)** o un **hiperplano (en dimensiones mayores)** que divida el espacio de datos en dos regiones, cada una correspondiente a una clase. El vector de pesos actúa como un vector normal a ese hiperplano, y su orientación y magnitud determinan la posición y dirección de la frontera de decisión.

Cuando un dato es clasificado incorrectamente, el modelo ajusta sus pesos desplazando el hiperplano en la dirección del error, lo que mejora su capacidad para separar correctamente los datos.

Componentes Clave de la Implementación en PyTorch

- `__init__`: Inicializa el vector de pesos en `1`.
- `get_weights`: Devuelve los pesos actuales.
- `run`: Calcula el producto escalar entre pesos y entrada.
- `get_prediction`: Aplica la función escalón y devuelve `+1` o `-1`.
- `train`: Realiza el entrenamiento hasta que no haya errores.
- Validación opcional: Permite evaluar el modelo en datos no vistos para estimar su rendimiento.

Resultados

El perceptrón binario demostró ser efectivo al clasificar correctamente los datos cuando estos son **linealmente separables**, como se aprecia en la gráfica adjunta. Durante el entrenamiento, la **línea negra** representa la frontera de decisión, la cual se va ajustando conforme se actualizan los pesos, separando progresivamente las dos clases de datos: `+1` (rojos) y `-1` (azules). Esta evolución visual evidencia cómo el modelo aprende y ajusta su frontera para lograr la mejor separación posible.

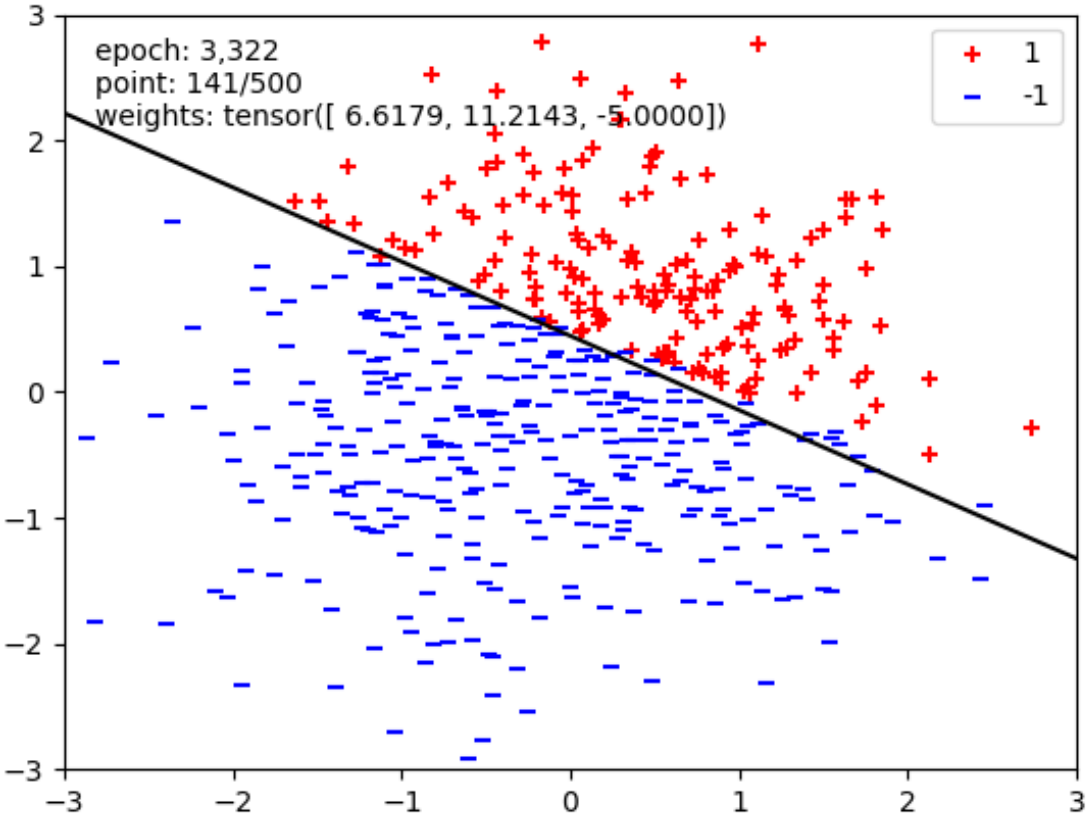

Figure 1: Entrenamiento Perceptrón Binario

Además, se observó que dividir el conjunto de datos en **entrenamiento y validación** acelera el proceso de convergencia. Esto se debe a que el conjunto de validación funciona como una guía para evaluar el rendimiento fuera de los ejemplos vistos, ayudando a detectar rápidamente si el modelo está sobreajustando o si aún necesita ajustes. A diferencia de entrenar con todos los datos mezclados, esta separación mejora la estabilidad del aprendizaje y permite una retroalimentación más inmediata sobre la calidad de los ajustes realizados.

Otro aspecto relevante es la importancia del **orden aleatorio (shuffle)** en los datos cuando se entrena con `batch_size=1`. Si los datos se presentan siempre en el mismo orden, es posible que el modelo entre en ciclos de error repetitivos que ralentizan o incluso impiden la convergencia. Al mezclar los datos en cada época, se evita este estancamiento, favoreciendo una separación más eficiente.

Finalmente, las pruebas realizadas sobre el conjunto de validación confirmaron el éxito del entrenamiento, mostrando que el modelo logró clasificar correctamente los datos según la frontera aprendida.

Problema 2 - Red Neuronal - Regresión no-lineal - Cálculo de la función sen(x)

Este problema aborda la implementación de una red neuronal feedforward (también conocida como red multicapa o MLP) que implementa un algoritmo de regresión no lineal para calcular la función sen(x) con x entre -2π y y 2π .

Objetivo del Modelo

El objetivo del modelo es recibir un valor x entre -2π y y 2π y devolver volver un valor y que representa el valor de x evaluado en la función seno. Para entrenar esta red, se usa el error medio cuadrado (mse_loss), ideal para modelos de regresión

Arquitectura del Modelo

El modelo está definido en la clase `RegressionModel`, y tiene la siguiente arquitectura:

- Capa de Entrada: 1 entrada y 150 salidas, activación RelU
- Capa Oculta 1: 150 entradas y 150 salidas, activación ReLU
- Capa de salida 150 entradas y 1 salida, sin activación final

```
super().__init__()
self.inp = Linear(1, 150)
```

```
self.layer = Linear(150, 150)
self.out = Linear(150, 1)
```

Funciones Principales

forward(x) Aplica la función de activación en la capa de entrada y en la capa oculta de la red neuronal, retorna la salida de la capa final de la red.

```
x = relu(self.inp(x))
x = relu(self.layer(x))
x = self.out(x)
return x
```

get_loss(x, y) Calcula la pérdida entre las predicciones de la red neuronal y los valores reales utilizando la función mse_loss.

```
predictions = self.forward(x)
return mse_loss(predictions, y)
```

train(dataset, epochs=...) Entrena el modelo usando descenso de gradiente con Adam (lr=0.001). El entrenamiento se ejecuta por varias épocas sobre el conjunto de entrenamiento. En cada época se calcula la pérdida total de cada batch en el dataset, luego, al final de iterar sobre cada batch se calcula la pérdida promedio, si esta es menor que 0.001 la iteración termina.

data = DataLoader(dataset, batch_size=70, shuffle=True) optimizer = optim.Adam(self.parameters(), lr=0.001) epochs = 2000

```
for epoch in range(epochs):
  total_loss = 0.0
  for batch in data:
    x_batch, y_batch = batch['x'], batch['label']
    optimizer.zero_grad()
    predictions = self(x_batch)
    loss = self.get_loss(x_batch, y_batch)
    loss.backward()
    optimizer.step()
    total_loss += loss.item()

  avg_loss = total_loss / len(data)
```

```
  if epoch % 200 == 0:
    print(f"Epoch {epoch}/{epochs} - Perdida: {avg_loss:.6f}")

if avg_loss <= 0.001:
    print(f"Deteniendo en la itreación {epoch} con perdida de {avg_loss:.6f}")
      break
```

Evaluación del Modelo

- Se calcula la pérdida total de cada batch y se mejoran los parámetros de la ecuación con el optimizador de Adam, luego de evaluar la pérdida en cada batch, se calcula la pérdida en promedio .
- Se espera que el modelo alcance al menos una pérdida menor a 0.02.

Resultados del modelo

- El modelo alcanza el valor de pérdida menor a 0.01 entre las 250 y 310 iteraciones en cada intento.
- La pérdida promedio oscila entre 0.0007 y 0.0008 en cada intento.

Problema 3 - Red Neuronal - Clasificación de Dígitos

Este problema aborda la implementación de una red neuronal feedforward (también conocida como red multicapa o MLP) entrenada para resolver el problema de clasificación de dígitos escritos a mano, utilizando el conjunto de datos MNIST. Cada imagen en este dataset es de 28×28 píxeles, lo que se transforma en vectores de 784 dimensiones.

Objetivo del Modelo

El objetivo del modelo es recibir una imagen (vector de 784 valores de tipo float) y devolver un vector de 10 dimensiones que representa los puntajes (logits) para cada clase del 0 al 9. El dígito más probable será el de mayor puntaje. Para entrenar esta red, se usa la pérdida de entropía cruzada (cross_entropy), ideal para tareas de clasificación multiclase.

Importante: No se debe aplicar activación ReLU en la última capa, ya que los logits deben pasar directamente a la función de pérdida.

Arquitectura del Modelo

El modelo está definido en la clase `DigitClassificationModel`, y tiene la siguiente arquitectura:

- Capa de Entrada: 784 neuronas (una por píxel)
- Capa Oculta 1: 150 neuronas, activación ReLU
- Capa Oculta 2: 50 neuronas, activación ReLU
- Capa Oculta 3: 50 neuronas, activación ReLU
- Capa de Salida: 10 neuronas (una por clase), sin activación final

```
self.fc1 = Linear(784, 150)
self.fc2 = Linear(150, 50)
self.fc3 = Linear(50, 50)
self.fc4 = Linear(50, 10)  # logits (sin ReLU aquí)
```

Funciones Principales

`run(x)` Ejecuta una forward pass por la red. Aplica activaciones ReLU en todas las capas excepto la última.

```
x = relu(self.fc1(x))
x = relu(self.fc2(x))
x = relu(self.fc3(x))
logits = self.fc4(x)
return logits
```

`get_loss(x, y)` Calcula la pérdida entre las predicciones (logits) y las etiquetas verdaderas y usando entropía cruzada.

`train(dataset, epochs=...)` Entrena el modelo usando descenso de gradiente con Adam (`lr=0.001`). El entrenamiento se ejecuta por varias épocas sobre el conjunto de entrenamiento.

Evaluación del Modelo

- Durante el entrenamiento, se calcula la precisión sobre el conjunto de validación con `dataset.get_validation_accuracy()`.
- Se espera que el modelo alcance al menos 97% de precisión en validación, aunque se recomienda apuntar a 97.5%-98% para asegurarse de pasar el calificador automático, que usa un conjunto de prueba oculto.

Buenas Prácticas de Entrenamiento

- Entrenar durante aproximadamente 5 épocas es suficiente en la mayoría de los casos.
- Si la precisión no mejora después de algunas épocas, puedes detener el entrenamiento antes.
- No aplicar softmax manualmente a la salida, ya que la función `cross_entropy()` se encarga internamente de eso.
- Asegurarse de que los datos estén bien normalizados (por ejemplo, escalados entre 0 y 1).