Motor de Redes Neuronales

Requisitos mínimos y opcionales

A continuación se detallan los requisitos mínimos que deben cumplir los proyectos para obtener una nota mínima de 5. El proyecto consiste en implementar desde cero (sin usar librerías de *deep learning*) un motor de redes neuronales que permita crear, entrenar y evaluar redes mediante *backpropagation* y un algoritmo de optimización. Al final se incluyen varias ideas opcionales que servirán para aumentar la nota.

1. Requisitos obligatorios

1.1. Implementación de redes con número variable de capas y neuronas

- Implementación de Redes Neuronales Densas (FNN)
- El motor debe aceptar arquitecturas completamente configurables, tanto en número de capas, como en número de neuronas por cada capa.
- Debe soportar, al menos, activaciones *sigmoid*.
- Debe manejar entradas de dimensiones arbitrarias (vectores) y producir salidas también en forma de vectores N-dimensionales.

1.2. *Forward Pass* y *Backpropagation*

- Implementación completa y correcta del *Forward Pass* y del *Backpropagation* para redes FNN.
- Cálculo correcto de gradientes para pesos y sesgos.
- Soporte de función de pérdida para *categorical cross-entropy* (en el caso de clasificación multi-clase) o MSE (regression).

1.3. Algoritmos de optimización

- Implementar el método de optimización Adam.
- Los optimizadores deben estar encapsulados en clases (ver 3.1) y exponer una API clara (por ejemplo, `step(params, grads)` o `update(params, grads)`).

1.4. Mini-batches de tamaño variable

- Entrenamiento por *mini-batches*: la implementación debe permitir un tamaño de batch ajustable por el usuario (p. ej. `batch_size=32, 64, 128, ...`).
- Deben manejarse correctamente los batches cuando el dataset no es divisible exactamente por el tamaño de batch.

1.5. División del dataset en training / validation / test

- Proporcionar funciones para dividir datos en train/val/test. Relaciones recomendadas: 60–80% train, 10–20% validation, 10–20% test.
- Los datos se deben poder mezclar de forma aleatoria.
- Debe existir la posibilidad de usar *random seed* para reproducibilidad de la partición.

1.6. Uso de *datasets* significativos

- El proyecto debe probarse sobre un *dataset* ampliamente utilizado que tenga un tamaño relativamente grande, como MNIST, Fashion-MNIST, etc.
- Se debe incluir instrucciones y/o código para descargar y preprocesar automáticamente estos *datasets* (normalización, codificación *one-hot* para etiquetas, *reshaping* si es necesario).

1.7. Separación lógica y modularización

- El proyecto debe estar bien modularizado: librería/paquete para la lógica (clases, funciones) y Jupyter Notebooks separados solo para pruebas y experimentos.
- El código del motor no debe mezclar código de entrenamiento/experimentos con la implementación de clases (ver 3.2 y 3.3).

1.8. Pruebas que verifiquen el correcto funcionamiento

- Deben realizarse pruebas automáticas y experimentos que demuestren que lo implementado funciona correctamente. Los entrenamientos deben mostrar gráficas de pérdida que demuestren que va disminuyendo y que la red aprende.

1.9. Restricciones de librerías

- Prohibido utilizar librerías de *deep learning* (PyTorch, TensorFlow, Keras, etc.).
- Solo se permiten librerías básicas y estándares, como *NumPy* para operaciones numéricas y *matplotlib*/*pandas* para visualización y manipulación de datos.

2. Estructura recomendada del repositorio

Se recomienda el uso de GitHub para el proyecto, aunque el código se debe entregar en un ZIP junto con la memoria. Este se debe modularizar convenientemente. Una estructura típica podría ser de la siguiente manera:

```
project-name/
├─ src/
│  ├─ network.py          # Clase NeuralNetwork y componentes básicos
│  ├─ layers.py           # Definiciones de capas, inicializaciones, activacion
│  ├─ optimizers.py       # Clases SGD, Adam
│  ├─ losses.py           # Funciones de pérdida
│  └─ utils.py            # Preprocesado, batching, split, one-hot
├─ tests/
│  ├─ test_gradcheck.ipynb
│  ├─ unit_tests.py
├─ notebooks/
│  ├─ demo_iris.ipynb
│  ├─ experiment_mnist.ipynb
├─ data/                  # scripts o instrucciones para descargar datasets
├─ README.md
└─ requirements.txt
```

Notas: - `src/` contiene todo el código reutilizable. Los *notebooks* en `notebooks/` son para pruebas y puesta a punto y no deben contener definiciones de clases principales. - `tests/` contiene *notebooks* o *scripts* que ejecutan pruebas sobre el código principal.

3. Detalles de diseño y API sugerida

A continuación se indican recomendaciones de diseño (no exhaustivas) para la implementación.

3.1. Clases mínimas requeridas

- `NeuralNetwork` (o `Network`): recibe una lista que describa la arquitectura y puede contener métodos como
  - `forward(x)`
  - `backward(loss_grad)` o `compute_gradients(x, y)`
  - `params()` → devuelve una lista/dict de parámetros (pesos y biases) para el optimizador
  - `zero_grad()` (opcionalmente limpiar gradientes)
- `Optimizer` (interfaz/base) y subclases concretas:
  - `Adam(lr, beta1, beta2, eps)` con método `update(params, grads)` (si se implementa)
  - `SGD(learning_rate, momentum=0.0, weight_decay=0.0)` con método `update(params, grads)`. (Esta método, y otros, son opcionales)

3.2. API de entrenamiento (ejemplo)

Se valorará positivamente un *Trainer* o una función `train` que gestione el bucle de entrenamiento:

```
trainer=Trainer(network, optimizer, loss_fn)

trainer.train(X_train, y_train, X_val, y_val, epochs=10, batch_size=64)
```

El *Trainer* debe: crear *mini-batches*, calcular *forward*, pérdida, *backward*, actualizar parámetros y calcular métricas en validación.

3.3. Inicialización de pesos

- Implementar inicialización de pesos razonable (Xavier/Glorot o He según la activación) o, en su defecto, inicialización aleatoria pequeña.

4. Pruebas obligatorias y criterios técnicos mínimos

Para alcanzar la nota 5, las pruebas y experimentos deben demostrar que la red empleada aprende y obtiene unos resultados mínimos aceptables. Por ejemplo, según el *dataset* que se elija se debe cumplir lo siguiente:

1. Entrenamiento en IRIS: la red debe converger y obtener una precisión razonable (recomendado >90% para una red simple). Si no se alcanza, el alumno debe incluir un análisis de por qué y cuáles son las limitaciones.
2. Entrenamiento en MNIST: demostrar que la red aprende. Para nota mínima se pide:
    - Que la pérdida de entrenamiento disminuya consistentemente durante las épocas.
    - Que la precision en *test* sea, al menos, ~80% (valor orientativo) si se usa una red razonable (dos capas ocultas) y entrenamiento básico. Si no se alcanza, el alumno debe justificar (implementación, hiperparámetros, tiempo de entrenamiento limitado, errores en preprocesado, etc.).

Nota: los porcentajes concretos pueden variar según la arquitectura y el *dataset* seleccionados. El requisito real es la evidencia clara de aprendizaje (curvas de pérdida, métricas, ejemplos clasificados) y no únicamente un número puntual sin explicación.

5. Extras y puntos adicionales

A continuación se detalla una lista, no exhaustiva, de otros temas opcionales que se pueden implementar para obtener una puntuación mayor que 5:

- Implementación de métodos de Optimización adicionales (SGD+Momentum, RMSProp, ...)
- Uso de varios *datasets* distintos
- Aplicación tanto a problemas de regresión como de clasificación, utilizando las funciones de pérdida adecuadas y realizando experimentos con *datasets* según el tipo de problema.
- Inclusión de varias funciones de activación distintas al *sigmoid* (*tanh*, ReLU, *softmax*, ...)
- Implementación de regularización: `weight_decay` (L2), *dropout*.
- Uso de *early stopping*.
- Implementación de varios *learning rate schedules* (*decay*, *step*, *cosine-annealing, etc.*)
- Realizar un estudio de los mejores hiperparámetros a seleccionar en los experimentos, comparándolos entre sí.
- Implementación de validación cruzada.