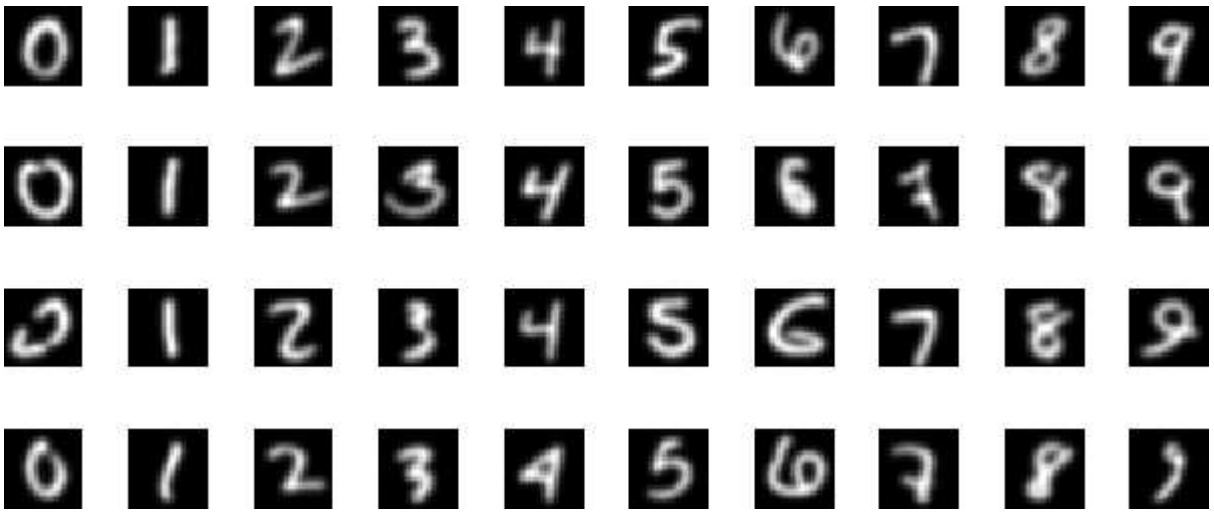
Proyecto Final 2025-1: AI Neural Network

CS2013 Programación III · Informe Final

Descripción



Este proyecto consistió en la implementación desde cero de una red neuronal multicapa en C++, orientada a resolver el problema de clasificación de dígitos manuscritos utilizando el dataset MNIST. El trabajo se enfocó en comprender a profundidad los fundamentos del aprendizaje profundo, desarrollando una solución funcional sin hacer uso de librerías de alto nivel como TensorFlow o PyTorch.

Contenidos

10. Licencia

Datos generales

- Tema: Redes Neuronales en AI
- Grupo: `GOGIS`
- Integrantes:
    - Nicolas Valentino Diaz Flores
    - Luis Enrique Cahuana Garcia
    - Gerard Deker Iruri Espinoza
    - Bruno Gonzalo Vega Napan

Requisitos e instalación

1. Compilador: GCC 11 o superior
2. Dependencias:
    - CMake 3.18+
3. Instalación:

```
git clone https://github.com/CS1103/projecto-final-gogi-gogiloni-c-hiper.git
cd projecto-final-gogi-gogiloni-c-hiper
mkdir build && cd build
cmake ..
Descargar archivos mnist_train.csv, mnist_test.csv, poner en carpeta mnist
```

4. `make`

---

1. Investigación teórica

Se investigaron los conceptos esenciales de redes neuronales, incluyendo su historia y evolución. Se estudiaron diferentes tipos de arquitecturas como las MLP, CNN y RNN, y se analizó a profundidad el algoritmo de backpropagation, así como el funcionamiento de optimizadores como SGD y Adam. Esta base teórica fue clave para guiar el desarrollo de la implementación.

---

2. Diseño e implementación

2.1 Arquitectura de la solución

El diseño del proyecto se basó en una estructura modular que permitió dividir el trabajo de manera eficiente. Se implementaron capas, funciones de activación, funciones de pérdida, optimizadores y el motor de entrenamiento. Se utilizaron principios de diseño como el patrón Strategy para permitir el intercambio flexible de optimizadores y funciones de pérdida.

Estructura del proyecto:

```
projecto-final-gogi-gogiloni-c-hiper/
├── .vscode/
├── build/
├── src/
│   └── utec/
│       ├── algebra/
│       │   └── tensor.h
│       └── neural_network/
│           ├── data/
│           ├── mnist_loader.h
│           ├── neural_network.h
│           ├── nn_activation.h
│           ├── nn_dense.h
│           ├── nn_interfaces.h
│           ├── nn_loss.h
│           └── nn_optimizer.h
│   ├── main.cpp
│   └── main2.cpp
├── tests/
├── .gitignore
├── CMakeLists.txt
├── README.md
├── main2.exe
├── modelo.nn
├── nn_train.exe
├── proyecto_final_test.exe
└── proyecto_final_train.exe
```

2.2 Manual de uso y casos de prueba

- Cómo ejecutar:

  g++ -O3 -march=native -std=c++17 -Iinclude -Isrc -o nn_train src/main.cpp

  g++ -O3 -march=native -std=c++17 -Iinclude -Isrc -o main2 src/main2.cpp

usamos optimizaciones porque demoraba mucho

- Casos de prueba:
    - Validación de inicialización de pesos y sesgos en capas Dense.
    - Pruebas unitarias de ReLU y Sigmoid.
    - Evaluación del descenso de la función de pérdida en 10 épocas.

---

3. Ejecución

Demo de ejemplo:
https://drive.google.com/drive/folders/1G-8KqxF7Qx92ONuX6BlmTdil2R7J3FR_

Durante la ejecución del programa, se siguió el siguiente flujo:

1. Carga de datos desde los archivos CSV (entrenamiento y prueba).
2. Normalización de las imágenes dividiendo los valores de píxel por 255.
3. Conversión de etiquetas numéricas a vectores one-hot.
4. Construcción de la red neuronal: 784 → 128 → ReLU → 64 → ReLU → 10 → Sigmoid.
5. Entrenamiento durante 100 épocas usando Adam como optimizador y BCELoss como función de pérdida.
6. Evaluación del rendimiento en el conjunto de prueba.

---

4. Análisis del rendimiento

- **Métricas de ejemplo:**
    - Iteraciones: 100 épocas.
    - Tiempo total de entrenamiento: 3150.46 seg
    - Precisión final: 96%.
- **Ventajas:**
    - Código ligero, limpio y modular.
    - Implementación propia del tipo Tensor, sin librerías externas.
- **Limitaciones:**
    - Sin paralelización ni uso de GPU.
    - Operaciones matriciales no optimizadas con BLAS o Eigen.
- **Mejoras futuras:**
    - Usar bibliotecas de álgebra optimizada como Eigen o BLAS.
    - Agregar soporte para entrenamiento en GPU o multihilo.

- Implementar almacenamiento y carga del modelo entrenado.

---

5. Trabajo en equipo

El desarrollo del proyecto fue colaborativo y cada integrante tuvo responsabilidades específicas, trabajando en conjunto para integrar los módulos de manera ordenada.

Tarea	Miembro	Rol
Investigación teórica	Nicolas Valentino Diaz Flores	Recolectó y resumió la base teórica de redes neuronales y optimizadores.
Diseño de la arquitectura	Luis Enrique Cahuana Garcia	Propuso la arquitectura de carpetas, nombres de clases, y organizó los módulos.
Implementación del modelo	Gerard Deker Iruri Espinoza	Codificó las capas (Dense, Activation), NeuralNetwork, y lógica de entrenamiento.
Entrenamiento, pruebas y demo	Bruno	Ejecutó los entrenamientos, evaluó precisión, ajustó hiper parámetros.

El código fue desarrollado en conjunto, realizando sesiones grupales para integrar módulos, resolver errores y validar resultados. La carga del dataset, el diseño de main.cpp, y las pruebas se realizaron en colaboración.

---

6. Conclusiones

Se logró implementar una red neuronal funcional desde cero, obteniendo resultados sólidos sobre el dataset MNIST. El trabajo permitió entender a fondo los conceptos de propagación hacia adelante, backpropagation y optimización de parámetros. Además, se logró modularizar el sistema de forma clara, lo que facilita futuras extensiones o mejoras. Se recomienda optimizar las operaciones algebraicas y explorar modelos más complejos en siguientes etapas.

---

7. Bibliografía

[1] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning, MIT Press, 2016.
[2] M. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.
[3] F. Chollet, Deep Learning with Python, Manning Publications, 2017.
[4] C. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

Licencia

Este proyecto usa la licencia MIT. Ver LICENSE para detalles.