Proyecto Final 2025-1: AI Neural Network
CS2013 Programación III · Informe Final

Contenidos

1. Descripción

Este proyecto consiste en el desarrollo de una red neuronal artificial desde cero utilizando C++, orientada a controlar la paleta de un jugador en el videojuego clásico Pong.
El juego Pong fue desarrollado usando SDL3 para gráficos y SDL_ttf para texto.
La IA controla una de las paletas, recibiendo como entrada el estado del juego (posición de la pelota, paddle, etc.).
La red neuronal procesa estas entradas y decide la acción óptima (subir, bajar o quedarse).
Se implementó un ciclo de entrenamiento donde la IA aprende a mejorar su desempeño a partir de la retroalimentación del entorno y poder adaptarse a la dinámica del juego en tiempo real. La red neuronal se entrena mediante el algoritmo de retropropagación y emplea funciones de activación.

2. Datos generales

Tema: Aplicación de Redes Neuronales en Inteligencia Artificial para Videojuegos

Integrantes

- De la Puente Rizo Patrón, Rodrigo – 202410045 (Implementación del modelo y pruebas)
- Paucar Barrios, Miguel Luis – 202410276 (Documentación y responsable de investigación teórica)

3. Requisitos e instalación

Compilador: GCC 11 o superior
Dependencias:

CMake 3.30+

- SDL3 (versión 3.2.16) para gráficos y eventos
- SDL_ttf (versión 3.2.2) para renderizado de texto
- Fuente OpenSans-Regular.ttf
- Compatible con C++20 o superior

Instalación:

- Clonar el repositorio y acceder a la carpeta del proyecto.
- Crear una carpeta de compilación y compilar con CMake:
- git clone https://github.com/CS1103/projecto-final-prograforever.git
- cd projecto-final-prograforever
- descarga externa de SDL y crear un model.dat para guardar los datos
- Ejecutar el binario generado (Play.exe, Trainer.exe).

4. Investigación teórica

En esta sección se busca explorar los fundamentos y arquitecturas de las redes neuronales artificiales (NN), comprendiendo su evolución, tipos principales y los algoritmos que permiten su entrenamiento.

4.1. Historia y evolución de las redes neuronales

Las redes neuronales artificiales surgieron inspiradas en el funcionamiento del cerebro humano. El primer modelo, el perceptrón, fue propuesto por Frank Rosenblatt en 1958, capaz de aprender tareas simples de clasificación. Sin embargo, sus limitaciones llevaron a un estancamiento hasta la década de 1980, cuando se desarrolló el algoritmo de retropropagación (backpropagation), permitiendo entrenar redes multicapa y resolver problemas más complejos.

En los años 2000 y 2010, el aumento del poder computacional y la disponibilidad de grandes volúmenes de datos impulsaron el auge del deep learning, con arquitecturas profundas capaces de superar a métodos tradicionales en tareas como visión por computadora y procesamiento de lenguaje natural.

4.2. Principales arquitecturas

- **MLP (Perceptrón Multicapa):**

Es la arquitectura más básica, compuesta por una capa de entrada, una o más capas ocultas y una capa de salida. Cada neurona de una capa está conectada a todas las neuronas de la siguiente (capas densas). Se utiliza para tareas de clasificación y regresión.

- **CNN (Redes Neuronales Convolucionales):**

Especializadas en el procesamiento de datos con estructura de grilla, como imágenes. Utilizan capas convolucionales que extraen características espaciales, seguidas de capas de pooling y densas. Son el estándar en visión por computadora.

- **RNN (Redes Neuronales Recurrentes):**

Diseñadas para procesar secuencias de datos (texto, audio, series temporales). Incorporan conexiones recurrentes que permiten mantener información de estados anteriores, siendo útiles en traducción automática, reconocimiento de voz, etc.

4.3. Algoritmos de entrenamiento

- **Backpropagation (Retropropagación):**

Es el algoritmo central para entrenar redes neuronales. Calcula el gradiente del error respecto a los pesos de la red, propagando el error desde la salida hacia las capas anteriores y ajustando los pesos para minimizar la función de pérdida.

- **Optimizadores:**

Los optimizadores determinan cómo se actualizan los pesos durante el entrenamiento. El método más simple es el descenso de gradiente (SGD), pero existen variantes avanzadas como Adam, RMSProp y Adagrad, que mejoran la velocidad y estabilidad del aprendizaje adaptando la tasa de aprendizaje.

5. Diseño e implementación

5.1. Arquitectura de la solución

5.1.1. Patrones de diseño utilizados

Patrón Singleton (implícito en SDL)

- Ubicación: Inicialización de SDL en el ciclo de vida del juego.
- Descripción: SDL gestiona recursos globales y su inicialización/destrucción se realiza una sola vez, siguiendo el principio Singleton para evitar múltiples instancias del subsistema gráfico.

Patrón Strategy

- Ubicación: En la red neuronal, especialmente en la selección de funciones de activación (nn_activation.h).
- Descripción: Permite intercambiar dinámicamente la función de activación (ReLU, Sigmoid, etc.) en las capas de la red, encapsulando el comportamiento y facilitando la extensión.

Patrón Serializador (Memento/Serialization)

- Ubicación: En model_serializer.h y model_serializer.cpp.
- Descripción: Permite guardar y restaurar el estado de la red neuronal (pesos y sesgos), encapsulando el proceso de serialización/deserialización y facilitando la persistencia del modelo.

Patrón Observer

- Ubicación: Al notificar sobre la carga en el proceso de entrenamiento, así como en la interacción entre el juego y la red neuronal.
- Descripción: Mientras se ejecuta el entrenamiento, se mandan los datos a una clase *TrainingProgressDisplay* que es atributo en *NeuralNetwork* y se encarga de imprimir los datos en pantalla. Por otro lado, en la interacción entre el juego y la red neuronal, el juego notifica a la red sobre el estado actual (posición de la pelota, paletas), y la red responde con una acción. Aunque esta última no está implementada como Observer de forma pura, la relación sigue el principio de notificación y reacción.

El uso de estos patrones aporta modularidad, flexibilidad y escalabilidad al proyecto. El patrón Strategy permite experimentar fácilmente con diferentes funciones de activación. El patrón Factory simplifica la creación y gestión de capas, mientras que la serialización asegura la persistencia y reutilización del modelo entrenado. La estructura modular facilita la extensión futura, como agregar nuevas capas o funciones de activación, y mejora el mantenimiento del código.

5.1.2. Estructura de carpetas

El proyecto está organizado en varias carpetas y archivos clave:

- -Play.exe : Abre la ventana del juego
- -Trainer.exe : Entrena y guarda los datos
- -src/: Carpeta con el código fuente principal.
- main.cpp: Punto de entrada, inicializa el juego y la red neuronal.
- Game.cpp/Game.h: Lógica del juego Pong, gestión de eventos y renderizado.
- Ball.cpp/Ball.h: Lógica de la pelota, colisiones y movimiento.
- Paddle.cpp/Paddle.h: Lógica de las raquetas.
- model_serializer.cpp/model_serializer.h: Serialización y deserialización del modelo de red neuronal: acceso a los archivos donde se almacenan los datos del proceso de entrenamiento.
- neural_network.h, nn_interfaces.h, nn_dense.h, nn_activation.h, nn_loss.h , nn_optimizer.h: Implementación de la red neuronal, capas densas y funciones de activación.
- tensor.h: Biblioteca genérica de álgebra donde se puede operar con tensores de N

dimensiones.
- training_observer.h: Implementa clases con métodos que permiten imprimir el proceso del entrenamiento.
- trainer.cpp: Archivo que ejecuta el proceso de entrenamiento.
- data/: Carpeta donde se almacena la data del modelo entrenado (model.dat).
- external/: Librerías externas (SDL3, SDL_ttf, fuentes).
- CMakeLists.txt: Configuración de compilación.

5.2. Manual de uso y pruebas realizadas

5.2.1. Manual de uso

Compilación

cd projecto-final-prograforeverl
run Play.exe o Trainer.exe (play para probar el juego funcionando, trainer para entrar el modelo)

5.2.2. Pruebas realizadas

No se implementaron casos de prueba específicos en este proyecto. Sin embargo, la mayoría de las características probables del juego se dan como salida en la consola. Por ejemplo, en cada época del entrenamiento se imprime la función de pérdida promedio de la época. Se puede observar que, con épocas suficientes y de suficiente tamaño, la función de pérdida decrece a lo largo del tiempo, lo que confirma el aprendizaje de la red neuronal. Además, se ha observado que modelos más entrenados logran mejores resultados en las partidas.

Tras terminar una partida con un modelo (salir del juego), se observa una estadística de en cuántos movimientos la red neuronal hizo una decisión acertada. En este caso, se usó como referencia si movía la raqueta en la dirección de la pelota, lo que también se usó a la hora de crear los datos para el entrenamiento. Sin embargo, este dato no es cien por ciento confiable, ya que, por ejemplo, la pelota podría moverse hacia arriba, rebotar y llegar al borde de la pantalla con una altura más baja de la que tenía originalmente, por lo que habría convenido moverse hacia abajo.

Por otro lado, se ha comprobado que es muy difícil hacer un punto con redes neuronales lo suficientemente entrenadas, mientras que es relativamente fácil hacerlo con redes neuronales poco entrenadas (esto depende de los parámetros colocados en el archivo del proceso de entrenamiento).

6. Ejecución

Para ejecutar el proyecto, antes que nada, hace falta compilar el proyecto siguiendo las instrucciones de instalación. De ahí, se puede correr los siguientes dos ejecutables.

Entrenamiento de la red neuronal

./Trainer.exe
Ruta: projecto-final-prograforever/Trainer.exe

Entrena la IA y guarda el modelo en data/model.dat.

Ejecución del juego con IA entrenada

./Play.exe
Ruta: projecto-final-prograforever/Play.exe

El juego se abrirá en una ventana.
Controles: W/S para mover la paleta izquierda, ESC para salir.

Para cambiar los parámetros del entrenamiento, se pueden editar los valores de los campos *training_data_size*, *epochs*, *batch_size* y *learning_rate*. Estos deben ser manejados con cuidado para evitar errores en la ejecución del entrenamiento o comportamientos no intencionales: por ejemplo, *learning_rate* no puede ser mucho más alto de su valor 0.01, mientras que *batch_size* no puede ser mayor que *training_data_size*, y tanto *training_data_size* como *epochs* deben tener valores lo suficientemente altos como para una buena ejecución del código (por ejemplo, mínimo 50 épocas y 5000 datos).

Demo del proyecto:

https://drive.google.com/file/d/1fqRw8XEEWMDnLs4SZF5vd3YAT8YNZqcO/view

Demo del código del proyecto:

https://drive.google.com/file/d/135XWxlW81_G_e4hfZrfN42Em7J7G1B0j/view

7. Análisis del rendimiento

Iteraciones de entrenamiento: Se realizaron múltiples épocas para ajustar los pesos de la red.
Tiempo de entrenamiento: Depende del hardware, pero se optimizó el código para minimizar latencias.

En general, la red neuronal logró aprender a anticipar la trayectoria de la pelota y posicionar la paleta de forma eficiente, sobre todo con procesos de entrenamiento más largos.

Ventajas generales

Código ligero y portable.
Sin dependencias de frameworks de ML externos.

Serialización eficiente del modelo.

Desventajas

Sin paralelización, el entrenamiento es secuencial.
Limitado a juegos simples por la arquitectura de la red.

Mejoras futuras

Implementar entrenamiento supervisado y basado en un sistema de recompensas por gol u otra razón parecida en vez de por posición de la raqueta.
Añadir optimizadores avanzados (Adam, RMSProp).
Paralelizar el entrenamiento para aprovechar CPUs multinúcleo.

8. Trabajo en equipo

TAREA	MIEMBRO	ROL
Implementación del modelo, diseño de la arquitectura del programa, pulido de la documentación	Rodrigo de la Puente	Esquemas de clases, código C++ de la red neuronal y del proyecto en general
Documentación e investigación	Miguel Paucar	Documentación, entrenado del modelo, pruebas, demo

9. Conclusiones

El proyecto permitió comprender y aplicar los fundamentos de las redes neuronales en un entorno práctico. Se logró implementar una red neuronal funcional en C++ y aplicarla con éxito al juego Pong, demostrando la capacidad de la IA para aprender comportamientos complejos a partir de datos simples.
El trabajo en equipo y la integración de diferentes módulos (juego, IA, serialización) fueron clave para alcanzar los objetivos propuestos. Se identificaron áreas de mejora para futuros desarrollos, como la optimización del entrenamiento y la ampliación a juegos más complejos.

10. Bibliografía

[1] S. Haykin, *Neural Networks and Learning Machines*. 3rd ed. Upper Saddle River, NJ: Pearson, 2009.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, NY: Springer, 2006.

[4] F. Chollet, *Deep Learning with Python*. Shelter Island, NY: Manning, 2017.

11. Licencia