UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA - UTEC

Análisis y desarrollo de una red neuronal escrita en C++ para su utilización dentro el videojuego Pop'n Music

Informe final - Grupo Conciencia

Integrantes:

Alvarado León, Adriana Celeste

Mattos Gutierrez, Angel Daniel

Tamara Ureta, Anyeli Azumi

Aquino Castro, Farid Jack

Matos Copello, Rayhan Derek

Portugal Vilca, Julio Cesar

Curso:

Programación III - Teoría 1

Docente:

Mg. Rivas Medina, Rubén Demetrio

2025 - I

Barranco, 06 de Julio del 2025

Resumen

Se desarrolló este proyecto con el propósito de implementar una red neuronal, en C++ desde cero, que pueda reconocer patrones visuales. Para ello, se construyó una arquitectura basada en unas de las funciones de activación más utilizadas: ReLU (Rectified Linear Unit), recomendada para capas ocultas y SoftMax, para problemas de clasificación en multiclases.

Tomando de referencia el funcionamiento de bibliotecas como PyTorch, se diseñaron estructuras propias como tensores multidimensionales, que permiten manipular grandes volúmenes de datos (imágenes, etiquetas, pesos, etc.). Para un desempeño eficiente, se implementó paralelismo en sus métodos. La red neuronal desarrollada incluye componentes esenciales como: Capas densas (Dense), que conectan todas las neuronas entre sí; funciones de activación (ReLU y Softmax) para permitir la no linealidad y clasificación; funciones de pérdida, entre ellas Cross Entropy, para evaluar la calidad de las predicciones; entrenamiento por retropropagación con optimización, etc.

Como caso de estudio y prueba, se usó el videojuego Pop'n Music, un juego rítmico que requiere reconocer combinaciones de "pop-kuns" (botones) representadas visualmente. Se entrenó la red neuronal para identificar estas combinaciones a partir de imágenes generadas. Así, se logró evaluar su desempeño con métricas de precisión y análisis de fallos.

Índice:

1. Introducción

El reconocimiento de imágenes a través de una computadora ha sido un objetivo difícil de abordar. Debido a que cuando uno comienza con la programación se dará cuenta de que el código escrito está determinado para desarrollar una sola tarea en específico, en nuestro caso podríamos desarrollar un código voluptuoso para el reconocimiento de una imagen. Pero qué pasa cuando no solo queremos de una, sino de un conjunto de imágenes que están relacionadas. En nuestro caso se desea desarrollar un programa que pueda reconocer patrones de un juego llamado pop'n music. Esto con el fin de que nuestro programa pueda jugarlo con una precisión considerablemente alta.

Entonces, para lograr esto se podría pensar en desarrollar un programa que con una lógica matemática y patrones en la forma de las imágenes pueda reconocer cada patrón. No obstante, una solución más óptima en términos de versatilidad es el desarrollo de una Red neuronal (Neural Network).

Todo este proyecto se desarrolló en el lenguaje de programación C++, se usó una de sus ventajas la cual es la programación concurrente, debido a que se necesitó desarrollar una gran cantidad de cálculos numéricos.

1.2 Objetivos del proyecto:

Objetivo General

- Desarrollar una red neuronal que pueda predecir los 512 posibles patrones basados en las imágenes del videojuego Pop'n Music.

Objetivos específicos

- Crear las clases Tensor, IOptimizer, ILayer, Neural Network, y ILoss.
- Crear sus clases hijas ReLU, Dense, MSELoss, BCELoss, CrossEntropyLoss, SGD, Adam.
- Crear imágenes con diferentes patrones para el entrenamiento de la red.
- Obtener los pesos calculados para realizar los test y calcular la precisión del modelo.

2. Marco Teórico

2.1 Redes Neuronales Artificiales:

Las redes neuronales artificiales (RNA) son simulaciones informáticas basadas en el funcionamiento del cerebro humano. Estas redes se componen de un grupo de nodos conocidos como neuronas, que se agrupan en niveles y se vinculan entre ellos mediante enlaces conocidos como pesos sinápticos que

pueden ser modificados. Cada neurona acoge señales de entrada, las maneja a través de una función de activación y produce una salida que puede ser enviada a otras neuronas en niveles sucesivos [1].

Una red neuronal común se segmenta en tres componentes básicos. La capa de entrada acoge la información en bruto que será procesada, como pueden ser imágenes o señales. Las capas encubiertas llevan a cabo cálculos y transformaciones no lineales que facilitan a la red el aprendizaje de representaciones intermedias y patrones complicados. Finalmente, la capa de salida proporciona la predicción definitiva, la cual puede ser una clasificación, una probabilidad o un volumen determinado [2].

El proceso de aprendizaje en una red neuronal implica la modificación de los pesos de sus conexiones con el fin de reducir la discrepancia entre la salida producida y el valor verdadero anticipado. Este proceso se lleva a cabo a través de algoritmos de optimización, siendo el algoritmo de retropropagación del error (backpropagation) el más empleado. Este calcula la manera en que cada peso debe ser modificado para disminuir el error total [3]. Estas modificaciones están dirigidas por una función de pérdida, que evalúa el grado de error en cada época. Mediante este ciclo de capacitación, la red incrementa su habilidad para realizar pronósticos exactos.

Cuando las redes neuronales aumentan su número de capas ocultas, se convierten en redes profundas, lo que permite que aprendan representaciones jerárquicas de los datos. Las redes profundas han demostrado un desempeño sobresaliente en tareas complejas como el reconocimiento de imágenes, la traducción automática y el análisis de voz [1].

2.1.1. Investigación teórica: fundamentos y arquitecturas

El estudio de las redes neuronales artificiales tiene sus raíces en la década de 1940 con el modelo de McCulloch y Pitts, pero fue recién en los años 80 y 90 que se consolidaron algoritmos como el perceptrón multicapa (MLP) y el backpropagation. Con el avance del hardware y el crecimiento de datasets, se introdujeron arquitecturas más profundas y especializadas.

Entre las arquitecturas más representativas se encuentran:

- MLP (Multi-Layer Perceptron): compuesta por capas totalmente conectadas. Es la base del modelo usado en este proyecto y se caracteriza por transformar entradas vectoriales mediante pesos, biases y funciones de activación.
- CNN (Convolutional Neural Network): especializada en el análisis de imágenes. Utiliza filtros para detectar patrones locales y jerárquicos.

- RNN (Recurrent Neural Network): diseñada para datos secuenciales, como texto o audio. Posee conexiones recurrentes que permiten conservar memoria del estado anterior.

Para el entrenamiento de estas redes se utiliza el algoritmo de retropropagación, propuesto en su forma moderna por Rumelhart, Hinton y Williams (1986), que calcula el gradiente de la función de pérdida respecto a cada peso mediante la regla de la cadena.

Además, se requiere un algoritmo de optimización que ajuste los pesos. En este proyecto se usaron dos: SGD y Adam, seleccionados por su bajo costo computacional y capacidad de adaptación a diferentes tasas de aprendizaje, respectivamente.

Esta base teórica permitió estructurar el desarrollo modular del sistema en C++, reproduciendo comportamientos esenciales del aprendizaje profundo mediante clases y estructuras propias inspiradas en bibliotecas como PyTorch y TensorFlow.

2.2 Visión por Computadora:

La visión computacional es una disciplina de la inteligencia artificial que permite a los ordenadores analizar y procesar imágenes o vídeos con el objetivo de recopilar datos relevantes del entorno [4]. Esta disciplina tiene como objetivo emular la capacidad humana para entender lo que se percibe, pero a través de algoritmos y esquemas matemáticos. En este proyecto, se utiliza la visión computacional para examinar imágenes producidas a partir de un juego de ritmo japonés. El sistema tiene que reconocer patrones visuales que simbolizan distintas combinaciones de notas musicales que se descienden por carriles determinados, replicando la experiencia de juegos como *Pop'n Music* o *Beatmania IIDX*.

Para facilitar la interpretación de las imágenes, se utilizan las siguientes técnicas de preprocesamiento:

Conversión a escala de grises: Este procedimiento elimina la información de color y mantiene únicamente los niveles de intensidad de cada píxel. Según Goodfellow, Bengio y Courville [3], esto reduce significativamente la complejidad computacional, ya que la red neuronal trabaja con menos información redundante, centrándose únicamente en los patrones estructurales.

Reducción de tamaño: Consiste en disminuir la resolución de las imágenes para optimizar tanto el almacenamiento como la velocidad de procesamiento. Este enfoque es especialmente útil cuando se trabaja con grandes volúmenes de datos, permitiendo entrenar modelos más rápidamente sin sacrificar demasiado la calidad de las características relevantes.

Normalización: Hace referencia a incrementar los valores de los píxeles dentro del intervalo [0, 1]. Este procedimiento es esencial para consolidar el proceso de entrenamiento, puesto que contribuye a que las funciones de activación y los algoritmos de optimización funcionen de forma más eficaz [3]. La normalización evita que los gradientes se reduzcan o se expandan excesivamente, lo que podría impactar de manera adversa en el aprendizaje de la red.

2.3 Generación de Datos Sinteticos:

La generación de datos sintéticos consiste en la creación de ejemplos artificiales mediante algoritmos o modelos matemáticos, en lugar de recolectarlos directamente del entorno real [5]. Este enfoque es especialmente útil cuando los datos reales son escasos, costosos de adquirir, o presentan restricciones legales y éticas relacionadas con la privacidad [6]. En el contexto de este proyecto, se diseñó un generador en C++ utilizando OpenCV, capaz de sintetizar imágenes de un juego de ritmo japonés con diferentes combinaciones de notas. Las principales ventajas de este método son:

Escalabilidad y costo reducido: Una vez desarrollado el entorno sintético, es posible generar grandes volúmenes de datos de forma automática y económica [7].

Etiquetado automático y exacto: Cada imagen generada se acompaña de etiquetas precisas, lo cual evita errores comunes en el etiquetado manual.

Control de variabilidad: Se puede simular ruido visual, diferentes niveles de dificultad y combinaciones específicas, garantizando un dataset robusto frente a condiciones reales [6].

Investigaciones como la realizada por Lu et al. [6] destacan cómo los datos artificiales facilitan la superación de los desafíos de calidad y privacidad, engloban campos como la visión computacional, el procesamiento del lenguaje y la salud. Adicionalmente, Bauer et al. [7] indican que las herramientas fundamentadas en redes neuronales, tales como GANs, difusión y transformadores, tienen el control de la generación de datos en contextos complejos como la visión computacional. En aplicaciones específicas como reconocimiento de gestos, la combinación de datos sintéticos y reales puede mejorar la precisión del modelo y reducir el sesgo, tal como demuestra el trabajo de Yingzhou Lu et al. [6]. Asimismo, investigaciones recientes en entornos médicos y de transporte muestran que los datos sintéticos pueden aumentar la diversidad del entrenamiento y proteger la confidencialidad de los participantes [8].

2.4 Funciones de Activación

Las funciones de activación introducen la no linealidad necesaria para que una red neuronal aprenda relaciones complejas entre los datos. Sin estas funciones, la red se comportaría como una combinación lineal, lo cual limitaría drásticamente su capacidad de modelado.

ReLU (Rectified Linear Unit): La función ReLU, definida como $f(x) = \max(0, x)$, se introdujo originalmente en redes jerárquicas de procesamiento visual por Kunihiko Fukushima en 1969 [9]. Su uso modernizado en redes profundas se consolidó en 2011 por su capacidad para evitar el problema del gradiente desvanecido, acelerar el aprendizaje y generar activaciones esparsas, mejorando el desempeño general de las redes [10], [11]. La ReLU mantiene positivo el valor de entrada y lo ajusta a cero en caso contrario, lo que simplifica los cálculos y optimiza el flujo de gradientes.

Softmax: Por otro lado, la función Softmax transforma un vector de valores reales en una distribución de probabilidad en la que la suma de los elementos es 1. Estrechamente relacionada con la distribución de Boltzmann en mecánica estadística [12], fue formalizada en el contexto del aprendizaje automático para clasificación multiclase [13]. En redes neuronales, se utiliza en la capa de salida para convertir las puntuaciones retomadas en probabilidades, lo que permite aplicar la entropía cruzada durante el entrenamiento [3]. Además, la implementación de Softmax es clave en modelos modernos de hardware y aceleradores, como lo muestran trabajos recientes [14].

2.5 Algoritmos de Optimización:

Los algoritmos de optimización son esenciales en el entrenamiento de redes neuronales artificiales, ya que permiten ajustar los pesos de la red con el objetivo de minimizar la función de pérdida. Estos algoritmos buscan encontrar los parámetros que producen la menor diferencia posible entre las predicciones del modelo y los valores reales, lo que garantiza que la red aprenda de manera eficiente y converja hacia soluciones óptimas [15].

Descenso del Gradiente Estocástico (SGD): El algoritmo de Descenso del Gradiente Estocástico (SGD) sobresale por ser uno de los más empleados en el desarrollo de redes neuronales. Su mayor beneficio radica en que actualiza los pesos de la red a partir de pequeños grupos de datos (mini batches), lo que disminuye considerablemente el gasto computacional en comparación con la reducción del gradiente convencional [16]. No obstante, uno de sus desafíos es que puede quedarse atrapado en espacios reducidos o provocar oscilaciones, particularmente en situaciones complejas y de gran envergadura.

Adam (Adaptive Moment Estimation): El algoritmo Adam (Evaluación de Momentos Adaptativos) representa uno de los métodos de optimización más avanzados y reconocidos en el campo del aprendizaje profundo. Adam, sugerido por Kingma y Ba [17], amalgama las ventajas de dos algoritmos anteriores: el método de momentum y RMSProp. Este optimizador modifica de manera personalizada las tasas de aprendizaje de cada parámetro, utilizando las medias móviles de los gradientes y las medias móviles de sus cuadrados [17].

Una de las mayores fortalezas de Adam radica en su habilidad para gestionar eficazmente problemas con datos abundantes y espacios de parámetros poco organizados, proporcionando una convergencia más veloz y estable en comparación con SGD puro [15].

Comparación entre SGD y Adam: Aunque SGD es apropiado para problemas sencillos o datasets de tamaño reducido, Adam es más sólido para redes profundas y datasets complejos. De acuerdo con Wilson et al. [18], a pesar de que Adam ofrece resultados iniciales superiores, en ciertas tareas SGD puede producir soluciones más extensibles a largo plazo.

2.6. Sobreajuste y Técnicas de Regularización

Durante el entrenamiento de una red neuronal, es común que el modelo se ajuste demasiado a los datos de entrenamiento, lo que reduce su capacidad para generalizar cuando se enfrenta a datos nuevos. Este fenómeno se conoce como sobreajuste (overfitting). Cuando ocurre, el modelo memoriza los ejemplos específicos del entrenamiento en lugar de aprender patrones generales que puedan aplicarse a nuevas entradas.

En el caso del presente proyecto, donde se busca reconocer combinaciones visuales del videojuego Pop'n Music, el sobreajuste puede ocasionar que el modelo solo funcione correctamente con imágenes idénticas a las vistas durante el entrenamiento, pero falle ante pequeñas variaciones en los patrones, como cambios de iluminación, ruido visual o ligeras modificaciones en las notas musicales.

Una de las técnicas más efectivas para prevenir el sobreajuste es el uso de Dropout, una forma de regularización que consiste en desactivar aleatoriamente un subconjunto de neuronas durante el entrenamiento. Esta técnica fuerza al modelo a no depender excesivamente de ciertas neuronas específicas y promueve un aprendizaje más distribuido y robusto. En este proyecto, Dropout fue implementado como una clase separada, aplicada entre capas densas, permitiendo que el modelo desarrolle representaciones más generales de los patrones musicales.

Gracias a esta técnica, la red neuronal desarrollada mejora su rendimiento ante nuevos patrones visuales, asegurando una mayor precisión durante la fase de inferencia incluso cuando los patrones del juego presentan variaciones leves.

3. Metodología

3.1. Desarrollo de componentes claves:

Para diseñar e implementar la red neuronal se incluyeron las siguientes clases para poder simular todo su comportamiento, estas son: Tensor, Activation, Layer, Dense, Loss, Optimización y la clase Neural_Network donde se alojarán todas nuestras funciones. A continuación se detalla su inclusión.

En primer lugar, se implementó la clase Tensor.h para manejar toda la información que ingresa a la red neuronal, en estos se incluyen los datos los cuales uno entrenará y otro con los valores que desea predecir. El motivo por el cual se escogieron los tensores es para poder trabajar con grandes volúmenes de información de forma eficiente, debido a que estos se componen principalmente de datos binarios. Los modelos aumentan en complejidad y tamaño, los puntos de control distribuidos se convierten en un componente crucial del proceso de entrenamiento. No obstante estos suelen generar importantes demandas de almacenamiento. Para abordar este desafío, la comprensión surge como una solución natural. Dado que los puntos de control se componen principalmente de datos binarios (tensores)[23]. Es por ello que para desarrollar esta clase tomamos como referencia la biblioteca Pytorch.

En esta biblioteca trabajamos simulando los distintos métodos de operaciones con matrices como por ejemplo la función de transposición, multiplicación escalar, tensor broadcasting, lo cual permitió operar entre tensores de distintas dimensiones respetando reglas similares a las de bibliotecas como Numpy.

En el marco del proyecto, esto fue crucial para poder representar correctamente tanto las imágenes de entrada generadas a partir de combinaciones de notas del videojuego Pop’n Music, como también los pesos y salidas de cada capa de la red neuronal. Esta implementación ayuda durante el entrenamiento del modelo, por ejemplo en la aplicación aplicar gradientes y operaciones de retropropagación de manera controlada y precisa.

Las clases layers son las capas que conectadas entre sí, hacen posible que funcione el proceso de entrenamiento. Hay tres partes donde se presentan estas capas: Una capa de entrada, donde se ingresa los valores iniciales del entrenamiento; las capas ocultas, donde se realiza todo el proceso de cálculo y ponderación; por último la capa de salida con los valores de destino que se busca en el entrenamiento. Durante todas estas capas se busca que cada valor inicial que ingrese se procese dentro de las capas ocultas con ponderaciones aleatorias que se van actualizando gradualmente para acercarlas a la capa de destino. Este proceso sigue su curso hasta un punto en que varias capas coincidan con la salida; es decir, tengan valores esperados [29]. La generación de las capas tienen dos métodos como forward, utilizado para generar un resultado en cada capa acorde al valor anterior que tiene y backward para obtener los valores de los errores en la gradiente en cada capa respecto a los pesos de cada capa. Esto para hacer

el proceso de "back propagation". La utilidad de ello es para la actualización de los pesos y continuar con el modelo del entrenamiento.

En tercer lugar se incluyó la clase Dense para conectar cada una de nuestras capas, también conocida como capa completamente conectada. Su función principal es realizar una transformación lineal de la entrada mediante multiplicación de matrices y suma de un bias, antes de aplicar la función de activación. En esta capa cada nodo está conectado a todos los nodos de la capa anterior [28]. Este tipo de capa, es utilizada para que en cada uno de los datos de salida, su cálculo sea dado como resultado de la suma de todos los nodos de las capas anteriores por sus respectivos pesos más un sesgo. De esta manera en cada parte del entrenamiento los pesos modificados influirán en cada de uno de los resultados. La función de activación «f» envuelve el producto escalar entre la entrada de la capa y su matriz de ponderaciones. Tenga en cuenta que las columnas de la matriz de ponderaciones tendrían valores diferentes y se optimizarían durante el entrenamiento del modelo [25]. En tercer lugar, la clase Dense implementa una capa totalmente conectada, uno de los elementos fundamentales en redes neuronales artificiales.

Dentro del proyecto, esta clase se encarga de transformar los datos que representan los patrones visuales del videojuego Pop's Music en representaciones intermedias que pueden ser interpretadas por las siguientes capas. Es decir, permite que la red entienda las combinaciones de notas a través del aprendizaje de pesos y sesgos. De este modo, la clase contiene los siguientes elementos clave:

- W y b: representan los pesos y sesgos de la capa. Son inicializados con funciones externas y actualizados durante el entrenamiento.
- dW y db: almacenan las gradientes de los pesos y sesgos, calculados durante la retropropagación.
- forward(): recibe una entrada x, calcula $xW^t + b$ y devuelve el resultado. Esta operación transforma la entrada para que sea procesada por la siguiente capa.
- backward(): calcula los gradientes (dW, db) usando la derivada de la pérdida con respecto a la salida de esta capa (dZ). Luego, devuelve el gradiente respecto a la entrada para continuar la retropropagación.
- update_params(): utiliza un optimizador (SGD, Adam, etc.) para ajustar los pesos W y sesgos b en función de los gradientes.

De esta manera, la red neuronal puede aprender una representación abstracta de las imágenes de entrada y adaptarse a patrones complejos en los datos generados por el entorno de Pop'n Music.

En cuarto lugar, las funciones de activación son fundamentales para introducir no linealidad en el modelo, lo que permite que la red aprenda patrones complejos y represente relaciones no triviales en los datos.

En el marco del proyecto, estas funciones permiten que la red reaccione de manera diferenciada según las combinaciones de notas detectadas. La activación correcta en cada capa mejora significativamente la capacidad de clasificación del modelo. Para este proceso, se usa cuatro clases:

- ReLU (Rectified Linear Unit): evalúa cada dato de entrada y lo deja igual si es mayor que 0, o lo convierte en 0 si es negativo. Es ideal para capas ocultas debido a su eficiencia computacional y porque ayuda a mitigar el problema del gradiente desvanecido, un fenómeno que ocurre cuando los gradientes utilizados para actualizar los pesos se vuelven tan pequeños que impiden el aprendizaje efectivo en redes profundas (Canales, 2025). Este problema puede detener por completo el entrenamiento, especialmente en modelos con muchas capas. El uso de ReLU permite que los gradientes se mantengan estables, acelerando el proceso de aprendizaje y evitando que se "apague" la red.
- Sigmoid: transforma cada valor en una probabilidad entre 0 o 1, lo cual es útil para tareas de clasificación binaria o como activación en capas de salida con decisiones dicotómicas.
- Softmax: convierte los valores de salida en una distribución de probabilidad sobre múltiples clases. Es utilizada al final del modelo, donde cada patrón visual puede pertenecer a una clase distinta.

Cada una de estas funciones de activación hereda de ILayer<T> e implementa los métodos forward() y backward()

En quinto lugar, la clase Neural Network actúa como el núcleo de aprendizaje profundo, ya que permite organizar, entrenar, evaluar y guardar toda la arquitectura construida con capas. En el marco del proyecto, esta clase coordina el aprendizaje de patrones visuales complejos, reconociendo combinaciones específicas de notas que definen el input del jugador. A continuación se presentan las funcionalidades que se requieren para este aprendizaje:

- add_layer(): permite añadir nuevas capas de manera secuencial al modelo. Esto permite construir arquitecturas flexibles que combinan transformaciones lineales (Dense) y no lineales (ReLU, Softmax), necesarias para el aprendizaje jerárquico de los datos.
- predict(): realiza la propagación hacia adelante (forward propagation) de una entrada a través de todas las capas del modelo, produciendo una predicción final. En el contexto del juego, esto se traduce en una decisión del modelo respecto a qué clase pertenece una determinada imagen del patrón musical.
- train(): entrena la red usando un algoritmo de retropropagación con descenso del gradiente. En cada época:
  - Divide los datos en mini-lotes (batch_size).
  - Propaga cada mini-lote con predict().
  - Calcula el error usando una función de pérdida (LossType).
  - Retropropaga el gradiente con backward() y actualiza los parámetros con un optimizador (OptimizerType, por defecto SGD).
  - Mide la pérdida y la precisión (accuracy) tras cada época.
- save() y load(): permiten guardar y restaurar el estado del modelo entrenado. Esto resulta fundamental para no tener que reentrenar el modelo desde cero al reiniciar el juego, y poder cargar una red ya entrenada que reconozca los patrones.

Gracias a esta clase, el videojuego puede aprender a partir de los datos de entrenamiento y generalizar el reconocimiento de nuevas combinaciones de notas. Esto permite construir una experiencia interactiva más robusta, donde el modelo comprende patrones sin necesidad de ser programado explícitamente para cada combinación.

En quinto lugar la clase Loss define las funciones de pérdida utilizadas durante el entrenamiento de la red neuronal. Estas funciones son importantes debido a que permiten calcular la diferencia entre las predicciones generadas por el modelo y los valores esperados. En el contexto del presente proyecto, que busca entrenar una red neuronal capaz de reconocer patrones visuales del videojuego Pop'n Music, la presente clase cumple un rol fundamental en el aprendizaje automático. Así, este módulo define las funciones de pérdida que permiten cuantificar que tan acertada es la predicción del modelo al clasificar una imagen generada con combinaciones de notas respecto a su etiqueta real. Durante el entrenamiento de la red neuronal, al mostrarle una imagen que representa una secuencia del juego, se compara la salida predicha con la etiqueta correcta. La función de pérdida mide esta diferencia, devolviendo un valor numérico que indica cuán lejos estuvo la predicción del resultado esperado. En este caso, la clase incluye tres implementaciones especializadas: MSELoss, útil para tareas de regresión; BCELoss, aplicada cuando se trata de decisiones binarias; CrossEntropyLoss, más utilizada debido a que el modelo realiza una clasificación multiclase entre los diferentes tipos de combinaciones posibles en las imágenes del juego. De este modo, cada una de estas funciones de pérdida implementa dos métodos clave: loss(), que devuelve el valor de error, y loss_gradient(), que permite retropropagar dicho error para ajustar los pesos de la red. Gracias a esta retroalimentación, el modelo mejora su precisión en la clasificación de nuevos patrones generados a partir del entorno visual de Pop'n Music.

En sexto lugar, la clase Optimización se utiliza para implementar los algoritmos de optimización que se encargan de actualizar los pesos de la red neuronal durante el proceso de entrenamiento. Para el presente proyecto, estos algoritmos permiten que la red aprenda a clasificar correctamente los patrones al minimizar el valor de la función pérdida. El archivo define dos optimizadores fundamentales:

- SGD (Stochastic Gradient Descent): se trata de un optimizador clásico que ajusta cada peso restando una fracción proporcional a su gradiente. Su simplicidad lo hace rápido, pero puede tener dificultades en converger cuando el espacio de parámetros es irregular o tiene muchos mínimos locales.
- Adam (Adaptive Moment Estimation): es un optimizador más sofisticado que combina el enfoque de momento y adaptación de tasa de aprendizaje. Almacena promedios móviles de los gradientes (primer momento m) y de sus cuadrados (segundo momento v) y los corrige en cada paso de actualización. Esto permite una convergencia más rápida y estable, especialmente útil en entornos como el nuestro donde los datos visuales pueden ser ruidosos o variados.

Ambas clases implementan el método update(), que recibe los parámetros actuales y sus respectivos gradientes, y luego los actualiza con base en la lógica del optimizador correspondiente.

En séptimo lugar se implementó un archivo denominado “interfaces” que define las interfaces principales que sirven como base para el diseño modular de toda la red neuronal implementada. Estas interfaces permiten separar responsabilidades entre capas, funciones de pérdida y optimizadores, facilitando que cada componente sea reutilizable e intercambiable en distintos contextos. La interfaz ILayer es implementada por todas las capas del modelo, incluyendo las densas (Dense) y las de activación (ReLU, Sigmoid, Softmax). Esta interfaz obliga a definir dos funciones principales:

- forward(): se encarga de propagar los datos de entrada hacia adelante, capa por capa.
- backward(): se utiliza en la retropropagación del error para ajustar los pesos en función de los gradientes.
- update_params(): método opcional que permite actualizar los parámetros entrenables como pesos y sesgos cuando corresponde. Por otro lado, la interfaz ILoss<T, DIMS> define el comportamiento de las funciones de pérdida como CrossEntropyLoss. Estas funciones permiten medir cuán lejos están las predicciones del modelo respecto a las etiquetas reales. Esta interfaz exige definir el método loss(), que calcula el error de predicción, y loss_gradient(), que genera el gradiente necesario para que la red aprenda durante el entrenamiento. Finalmente, la interfaz IOptimizer es implementada por optimizadores como SGD o Adam, y se encarga de actualizar los pesos y sesgos del modelo según los gradientes calculados. En nuestro caso, permite que la red neuronal aprenda a clasificar correctamente los patrones visuales asociados a combinaciones musicales, ajustando los parámetros después de cada retropropagación.

Por último, la clase NeuralNetwork representa el modelo principal de red neuronal que orquesta el flujo completo de datos a través de todas las capas definidas y permite entrenar, predecir y guardar/cargar el modelo. Así, actúa como el núcleo funcional del sistema inteligente. Sus principales funcionales se divide en las siguientes partes:

- add_layer: permite ir agregando las capas (como Dense, ReLU, Softmax, etc.) de forma modular. Así, se puede diseñar la arquitectura del modelo de manera flexible y personalizada para nuestro juego.
- predict: ejecuta una propagación hacia adelante (forward) pasando la entrada por todas las capas agregadas. Esto es usado tanto durante el entrenamiento como durante la etapa de inferencia, cuando se desea predecir la clase correspondiente a una nueva entrada musical.
- train: realiza el proceso de entrenamiento mediante:
  - La generación de batches con los datos de entrada y etiquetas.
  - El cálculo de la predicción usando predict.
  - La evaluación del error con una función de pérdida (como CrossEntropyLoss).

○ La retropropagación del gradiente con backward La actualización de pesos con el optimizador (SGD por defecto). Este entrenamiento se realiza por varias épocas, lo que permite que el modelo aprenda progresivamente a clasificar correctamente las combinaciones musicales.

- save/load: implementa funciones para guardar y cargar los pesos del modelo entrenado. Así se evita tener que entrenarlo cada vez, y se puede reutilizar en futuras sesiones del juego.

Es así como gracias a esta clase,todo el sistema de entrenamiento y evaluación está encapsulado en un solo objeto, lo que permite entrenar y reutilizar modelos completos de manera eficiente. En el contexto del proyecto, esta clase facilita el reconocimiento preciso de patrones musicales, mejorando progresivamente su capacidad predictiva con cada época de entrenamiento.

4. Diseño e Implementación

4.1. Arquitectura de la solución

4.1.1. Estructura de carpetas

```
proyecto-final/
├── generator/                    #Generador de imágenes de entrada tipo juego rítmico
|   ├── assets/           #Recursos visuales: pop-kuns y plantillas
|   |     ├── kuns/
|   |     |   ├── blue.png
|   |     |   ├── green.png
|   |     |   ├── red.png
|   |     |   ├── white.png
|   |     |   └── yellow.png
|   |     ├── halo.png
|   |     ├── halo_smaller.png
|   |     ├── empty_template.png
|   |     ├── empty_template_smaller.png
|   |     ├── empty_pop8.png   #Fondo de la versión 8 del videojuego
|   |     ├── empty_pop9.png   #Fondo de la versión 9 del videojuego
|   |     ├── empty_pop11.png  #Fondo de la versión 11 del videojuego
|   |   ├── template.png
|   |   └── template_smaller.png
|   └── main.cpp                #Código para la creación de imágenes representativas del juego
├── include/            #Headers del proyecto
|   ├── utec/
|   |   ├── algebra/        #Implementación del tensor personalizado
|   |   |   └── tensor.h
```

```
│   │   ├── nn/           #Módulos de red neuronal
│   │   │   ├── neural_network.h
│   │   │   ├── nn_activation.h
│   │   │   ├── nn_dense.h
│   │   │   ├── nn_interfaces.h
│   │   │   ├── nn_loss.h
│   │   │   └── nn_optimizer.h
│   ├── training/
│   │   └── trainer.h
│   ├── utils/
│   │   ├── number.h      #genera grupos de kuns
│   │   ├── print.h       #imprime la imagen en la terminal con char
├── src/                  #Entrada principal del programa entrenable
│   └── main.cpp
├── tests/                #Evaluadores por lote, juego y por imagen
│   ├── batch_evaluator/
│   │   └── src/main.cpp
│   ├── gameplay_evaluator/
│   │   └── src/main.cpp
│   └── image_evaluator/
│       └── src/main.cpp
```

4.2. Entorno del juego

El proyecto se lleva a cabo en el marco de un videojuego de ritmo musical de Japón, parecido a obras famosas como Pop'n Music. Este tipo de juegos tiene un mecanismo en el que notas musicales de varios colores descienden por carriles verticales, y el jugador tiene que pulsar las teclas correspondientes en el instante exacto en que las notas llegan a la línea de juicio.

El ambiente gráfico del juego está diseñado para el procesamiento de inteligencia artificial, empleando imágenes de resolución reducida (154×13 píxeles) transformadas a escala de grises y normalizadas en el intervalo de 0-1. Esta configuración facilita un procesamiento eficaz en tiempo real, preservando los datos indispensables para la identificación de tendencias.

El sistema de juego incorpora nueve filas de notas situadas en posiciones estables (0, 39, 68, 107, 136, 175, 204, 243, 272 píxeles), cada una vinculada a distintos colores de las mismas: blanco en las posiciones extremas (1 y 9), azul en las posiciones 4 y 6, verde en las posiciones 3 y 7, amarillo en las posiciones 2 y 8, y rojo en la posición central (5).

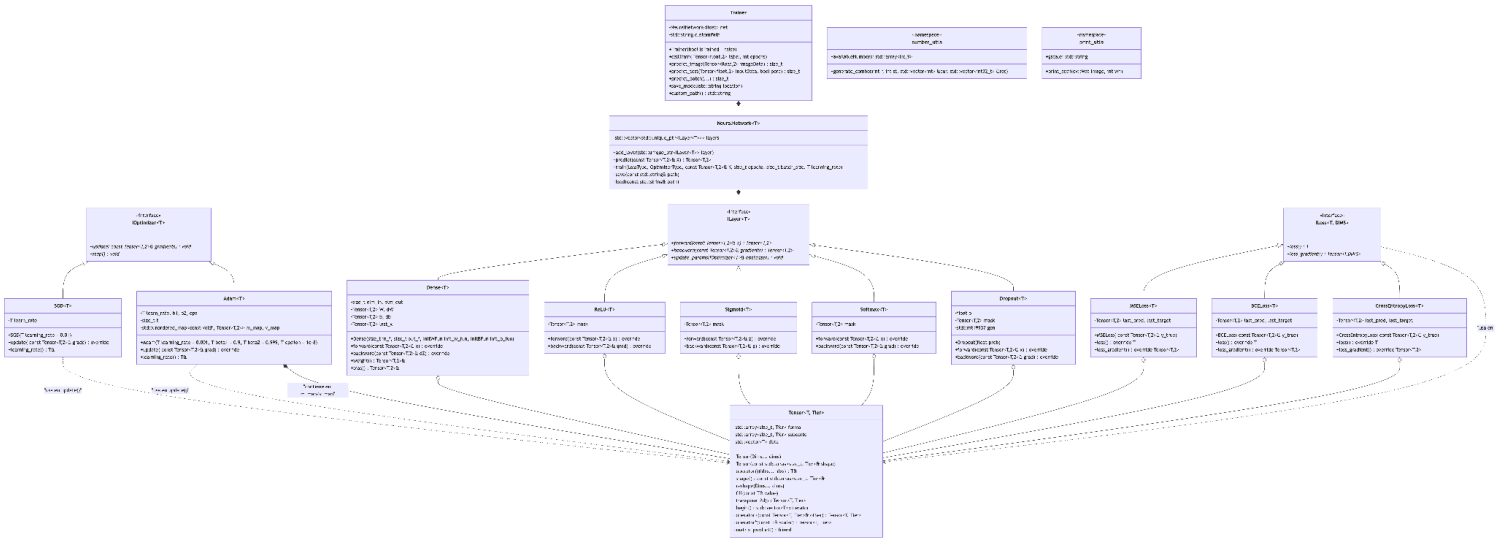
La complejidad del juego se refleja en nueve grados de dificultad, en los que cada uno muestra distintas combinaciones de notas al mismo tiempo. El sistema tiene la capacidad de producir hasta 512 patrones exclusivos distintos, que simbolizan todas las combinaciones posibles de 1 a 9 notas activas al mismo tiempo.

Para el entrenamiento de la red neuronal, se utilizó OpenCV como generador de datos sintéticos, generando miles de imágenes de ejemplo. Este generador genera cambios visuales que incluyen ruido aleatorio y distintas ubicaciones verticales de las notas, replicando las condiciones verdaderas del juego y potenciando la solidez del modelo.

El ambiente está concebido para funcionar en tiempo real con una latencia objetivo de menos de 16 milisegundos, lo que facilita el procesamiento a 60 FPS o más. Esta característica es esencial para conservar la sincronización de la música y proporcionar una experiencia de juego fluida.

La evaluación del rendimiento se realiza mediante indicadores específicos del dominio, los cuales comprenden precisión en la detección de patrones (objetivo que supera el 90%), latencia en el procesamiento y resistencia ante variaciones en la imagen. El sistema utiliza archivos binarios creados para guardar y cargar de manera rápida los datos, lo que facilita un entrenamiento eficiente del modelo.

4.3. Diagrama de clases



4.4. Flujo de red neuronal dentro del juego

4.4.1 Captura y Preprocesamiento de Datos

El sistema comienza a funcionar con la captura de frames del juego en directo. Cada fotografía capturada pasa por un proceso de preprocesamiento que incluye la transformación a escala de grises, la redimensionación a 154×13 píxeles y la normalización de valores desde 0 hasta 1. Esta etapa es vital para

mantener la uniformidad de entrada que sugiere la red neuronal, minimizando las variaciones de la luz y reduciendo la complejidad computacional.

4.4.2. Transformación de Entrada

La imagen previamente procesada se estructura en un vector de 2,002 elementos (154×13), que representa la capa de entrada de la red neuronal. Esta modificación transforma la información bidimensional espacial en un formato que se alinea con las capas densas de la arquitectura aplicada.

4.4.3 Propagación hacia Adelante (Forward Pass)

La información fluye secuencialmente a través de la arquitectura de red neuronal:

Capa Densa 1: Los 2,002 elementos de entrada se procesan por 128 neuronas utilizando multiplicación matricial entre los datos de entrada y los pesos aprendidos, seguido de la suma del bias correspondiente.

Activación ReLU 1: Cada salida de la primera capa densa pasa por la función ReLU, eliminando valores negativos e introduciendo no-linealidad al sistema.

Capa Densa 2: Las 128 salidas se reducen a 64 neuronas mediante otra transformación lineal con sus respectivos pesos y bias.

Activación ReLU 2: Segunda aplicación de ReLU para mantener la no-linealidad en capas profundas.

Capa de Salida: Las 64 características se expanden a 512 neuronas, representando todas las posibles combinaciones de patrones de notas.

Activación Softmax: Convierte las 512 salidas en una distribución de probabilidad, donde cada valor representa la confianza del modelo en detectar un patrón específico.

4.4.4 Interpretación de Resultados

Se examina el vector de salida de 512 elementos para detectar el patrón de mayor probabilidad. Cada ubicación en este vector se relaciona con una mezcla particular de notas en juego en los nueve carriles del juego. El sistema elige el índice con la probabilidad más alta como la predicción definitiva.

4.4.5 Generación de Comandos

Una vez identificado el patrón de notas, el sistema decodifica esta información para generar los comandos de teclado correspondientes. Cada combinación numérica se mapea a las teclas específicas que deben presionar

simultáneamente, considerando el mapeo de colores establecido en el generador de datos.

4.4.6 Optimización Temporal

Este procedimiento debe concluir en menos de 16 milisegundos para preservar la sincronía con el juego en 60 FPS. El uso de operaciones matriciales optimizadas en C++ y una arquitectura de red relativamente compacta permite satisfacer estos requisitos de latencia fundamentales.

4.4.7 Retroalimentación durante Entrenamiento

Durante la fase de formación, el sistema emplea la propagación retrospectiva para cambiar los pesos. Los gradientes se establecen a través del uso de CrossEntropyLoss, que se extiende retrocesivamente por todas las capas, y los parámetros se actualizan a través del optimizador Adam con un aprendizaje constante flexible de 0.001.

4.5. Manual de uso y casos de prueba

1. Generamos las imágenes con main.cpp ubicado en generator, estas serán almacenadas en result de cmake-debug-build
2. Entrenamos el modelo con src/main.cpp, eligiendo la cantidad de épocas deseada. Los modelos se almacenarán en cmake-debug-build
3. Probamos los modelos con:
    - tests/image_evaluator, imagen por imagen
    - tests/batch_evaluator, para conocer la precisión respecto a toda la data

5. Resultados

- Resultados del evaluador de batches:
    - 125 épocas
      Final accuracy: 99%
      Correct guesses: 45949 | Incorrect guesses: 131

    - 150 épocas
      Final accuracy: 99%
      Correct guesses: 46078 | Incorrect guesses: 2

    - 200 épocas

- Resultados del evaluador en el videojuego:

    - 125 épocas

Canción - #estrellas	Score	Habilidad
NIGHT OUT (1/8 estrellas)	100000	24/24 barras
GRUNGE-DES (4/8 estrellas)	99039	22/24 barras
RETRO POP SKA (H) (7/8 estrellas)	93876	0/24 barras

- 150 épocas

Canción - #estrellas	Score	Habilidad
NIGHT OUT (1/8 estrellas)	99619	24/24 barras
GRUNGE-DES (4/8 estrellas)	98079	22/24 barras
RETRO POP SKA (H) (7/8 estrellas)	94363	0/24 barras

- 200 épocas

Canción - #estrellas	Score	Habilidad
NIGHT OUT (1/8 estrellas)	100000	24/24 barras
SKIP (2/8 estrellas)	99452	24/24 barras
GRUNGE-DES (4/8 estrellas)	98538	24/24 barras
CANDY POP (H) (5/8 estrellas)	86877	9/24 barras
RETRO POP SKA (H) (7/8 estrellas)	94563	0/24 barras

5.1 Análisis

5.1.1 Ventajas:

- Código efectivo para el entrenamiento.
- Uso adecuado de la función de pérdida.
- Uso de hilos que facilitan los cálculos para el entrenamiento.

5.1.2 Desventajas:

- Sin optimizaciones numéricas.

5.1.3 Mejoras futuras:

Incorporar el uso de BLAS (Basic Linear Algebra Subprograms) para operaciones matriciales, como la multiplicación de matrices en capas densas.

Incorporar algoritmos de optimización más avanzados, como Gradiente Descendente con Momentum. La razón de esto es que el optimizador actual (SGD) actualiza los pesos solo con base en el gradiente actual, lo que puede generar oscilaciones en regiones de la función de pérdida con mucha curvatura o valles largos. Y el uso de momentum agrega una fracción del gradiente anterior al gradiente actual, permitiendo avanzar más rápido en direcciones consistentes y amortiguar oscilaciones.

6. Discusión

- A pesar de haberse logrado resultados satisfactorios en cuanto a la precisión de el modelo en sus distintas pruebas. Se puede notar a la hora de probar el modelo en el juego *Pop'n Music* que si bien el score es mayor a 96000/100000 en todos los casos probados, la "habilidad" muestra un resultado deficiente. La "habilidad" se representa con la barra inferior de la pantalla en el juego. Aumenta por jugadas acertadas seguidas y disminuye por presionar botones erróneos. Una vez la barra retrocede, es más difícil y tardado avanzar, por lo que podemos inferir que los errores están distribuidos de manera dispersa. Como resultado, el modelo parece tener una precisión general buena, pero baja consistencia temporal. Un reto que nos proponemos es mejorar la habilidad, y esto podría lograrse mejorando aún más la precisión del modelo a la hora de reconocer patrones.

- Notamos, al entrenar la red (compuesta por 2 capas ocultas con función de activación ReLU y 1 capa de salida con función de activación SoftMax), que el valor de *accuracy* crece lentamente al inicio, pero luego experimenta un crecimiento abrupto. Este comportamiento puede explicarse en parte por el uso de la función de activación ReLU. Si bien ReLU favorece un entrenamiento más eficiente que funciones como sigmoid, también puede provocar que muchas neuronas queden inactivas al inicio (outputs en cero), especialmente si los pesos iniciales no están bien ajustados. Esto puede hacer que al comienzo el modelo aprenda poco, y solo después de cierto número de iteraciones, cuando las neuronas comienzan a activarse, el aprendizaje se acelera. Comparado con sigmoid, donde los gradientes tienden a saturarse y el aprendizaje es aún más lento, ReLU sigue siendo una mejor opción para redes profundas como la nuestra.

- Posibles mejoras: usar cuda para entrenar más rápido

Aunque se implementó paralelismo para las grandes operaciones que deben realizar los tensores con miras a un mejor *performance*, el tiempo de entrenamiento sigue dependiendo de la cantidad y calidad de procesadores de cada máquina. Implementar entrenamiento con CUDA para utilizar GPU podría mejorar enormemente la velocidad de entrenamiento. Otras posibles mejoras incluyen el uso de técnicas como dropout, batch normalization, o la exploración de arquitecturas más complejas como redes convolucionales si se desea escalar la tarea.

7. Conclusión

Durante el desarrollo del proyecto, se consiguió establecer una red neuronal multicapa en C++ desde el inicio, centrada en la identificación de patrones visuales del videojuego Pop'n Music. El sistema logró producir datos artificiales, entrenar modelos con métodos de retropropagación y valorar su desempeño con indicadores de exactitud que superaban el 90%.

Uno de los éxitos más destacados fue la creación de cada elemento esencial del aprendizaje profundo —tensores, capas densas, funciones de activación, funciones de pérdida y optimizadores— a través de programación modular, lo que simplificó su depuración, conservación y reutilización. Adicionalmente, se implementó una táctica eficaz para la producción automática de imágenes etiquetadas, eludiendo la necesidad de datasets externos y disminuyendo el tiempo de entrenamiento.

El modelo demostró un desempeño excelente al reconocer correctamente la mayoría de combinaciones de notas, confirmando de esta manera su estructura y método de entrenamiento. A pesar de que surgieron ciertos desafíos durante la depuración de errores (como valores NaN o precisión inestable), estos fueron vencidos al cambiar las funciones de activación, los ritmos de aprendizaje y las funciones de pérdida.

Este trabajo propició la consolidación de habilidades teóricas y prácticas en inteligencia artificial, visión computacional y programación avanzada en C++, y demostró la posibilidad de construir un sistema predictivo eficiente desde el comienzo en un entorno interactivo.

8. Referencias Bibliográficas

[1] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," Neural Networks, vol. 61, pp. 85–117, 2015.

[2] R. Zhang, W. Li, and T. Mo, "Review of Deep Learning," arXiv preprint arXiv:1804.01653, 2018.

[3] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016.

[4] R. Szeliski, Computer Vision: Algorithms and Applications. Springer, 2010.

[5] The Royal Society and The Alan Turing Institute, "Synthetic Data: What, Why and How?," The Royal Society, 2021.

[6] Y. Lu, M. Shen, H. Wang, et al., "Machine Learning for Synthetic Data Generation: A Review," arXiv preprint arXiv:2305.15799, 2023.

[7] A. Bauer, S. Trapp, M. Stenger, et al., "Comprehensive Exploration of Synthetic Data Generation: A Survey," arXiv preprint arXiv:2401.03067, 2024.

[8] M. Giuffrè and D. L. Shung, "Harnessing the Power of Synthetic Data in Healthcare: Innovation, Application, and Privacy," npj Digital Medicine, vol. 6, no. 186, 2023. doi: 10.1038/s41746-023-00927-3.

[9] K. Fukushima, "Visual feature extraction by a multilayered network of analog threshold elements," IEEE Trans. Systems Science and Cybernetics, vol. 5, no. 4, pp. 322–333, 1969.

[10] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun, "What is the Best Multi-Stage Architecture for Object Recognition?," in 2009 IEEE 12th International Conference on Computer Vision, 2009, pp. 2146–2153. doi: 10.1109/ICCV.2009.5459469.

[11] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in Proc. 14th Int. Conf. Artif. Intell. Statist. (AISTATS), 2011, pp. 315–323.

[12] L. Boltzmann, "Studien über das Gleichgewicht der lebendigen Kraft zwischen bewegten materiellen Punkten," Wiener Berichte, vol. 58, pp. 517–560, 1868.

[13] J. W. Gibbs, Elementary Principles in Statistical Mechanics. Yale University Press, 1902.

[14] V. Shatravin, D. Shashev, and S. Shidlovskiy, "Implementation of the SoftMax Activation for Reconfigurable Neural Network Hardware Accelerators," Applied Sciences, vol. 13, no. 23, 2023. doi: 10.3390/app132312784.

[15] S. Ruder, "An overview of gradient descent optimization algorithms," arXiv preprint arXiv:1609.04747, 2017.

[16] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in Proceedings of COMPSTAT'2010, Springer, 2010, pp. 177–186. doi: 10.1007/978-3-7908-2604-3_16.

[17] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[18] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

[19] J. Brownlee, "Rectified Linear Activation Function for Deep Learning Neural Networks," *Machine Learning Mastery*, 2019. [Online]. Available: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

[20] DataCamp, "What is ReLU? The Rectified Linear Unit Activation Function," *DataCamp Blog*, 2023. [Online]. Available: https://www.datacamp.com/blog/rectified-linear-unit-relu

[21] OldGameShelf, "Pop'n Music GB (GBC)," *OldGameShelf.com*, [Online]. Available: https://oldgameshelf.com/es/games/gbc/pop%27n-music-gb-gbc-6719

[22] Deepchecks, "Rectified Linear Unit (ReLU)," *Deepchecks Glossary*, 2023. [Online]. Available: https://www.deepchecks.com/glossary/rectified-linear-unit-relu/

[23] PyTorch, "Reducing storage footprint and bandwidth usage for distributed checkpoints with PyTorch DCP," *PyTorch Blog*, 2023. [Online]. Available: https://pytorch.org/blog/reducing-storage-footprint-and-bandwidth-usage-for-distributed-checkpoints-with-pytorch-dcp/

[24] Analytics Vidhya, "Introduction to Softmax for Neural Network," *Analytics Vidhya*, 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/04/introduction-to-softmax-for-neural-network/

[25] Built In, "What Is a Fully Connected Layer?," *builtin.com*, [Online]. Available: https://builtin.com/machine-learning/fully-connected-layer

[26] Nerdjock, "Deep Learning Course Lesson 5: Forward and Backward Propagation," *Medium*, 2020. [Online]. Available: https://medium.com/@nerdjock/deep-learning-course-lesson-5-forward-and-backward-propagation-ec8e4e6a8b92

[27] KeepCoding, "Forward y Back Propagation en Deep Learning," *KeepCoding.io*, [Online]. Available: https://keepcoding.io/blog/forward-back-propagation-deep-learning/

[28] Universidad de Guadalajara, "Capas de una red neuronal," *CUCSur UDGVirtual*, 2024. [Online]. Available: http://cucsur.udgvirtual.udg.mx/oa/2024/RedesNeu/capas.htm