Universidad de Buenos Aires

Maestría en Explotación de Datos y Descubrimiento de Conocimiento

Neural Architecture Search con Aprendizaje por Refuerzo

Aprendizaje Reforzado

Autor: Esp. Lic. Tschopp Joaquín Sebastian

Docentes: Dr. Ing. Cesar Caiafa

CABA, Argentina
5 de diciembre de 2025

Resumen

Este trabajo investiga la Búsqueda de Arquitectura Neuronal (NAS) mediante Aprendizaje por Refuerzo, replicando y analizando la metodología de Zoph y Le (2016) para desafiar la complejidad del diseño manual de redes profundas. El estudio se estructura en dos fases experimentales sobre el dataset CIFAR-10: la ejecución de una búsqueda automatizada bajo estrictas restricciones computacionales y la reconstrucción forense de la arquitectura NASCNN15, cuyos detalles de implementación originales son escasos. Los resultados validan la eficacia del diseño algorítmico frente a la intuición humana. Primero, la búsqueda automatizada logró descubrir una arquitectura no convencional de 13 capas y apenas 275,000 parámetros que maximizó la recompensa en un entorno limitado. Segundo, la reproducción de la arquitectura NASCNN15 alcanzó una exactitud del 91.06 % en el conjunto de prueba, demostrando un rendimiento competitivo frente a modelos con mayor densidad de parámetros. Se concluye que la optimización automatizada mediante RL es capaz de descubrir topologías eficientes y "livianas"que, pese a su menor complejidad estructural, rivalizan en desempeño con arquitecturas modernas significativamente más costosas.

Palabras clave: Búsqueda de Arquitectura Neuronal, Aprendizaje por Refuerzo, CIFAR-10, Deep Learning, Optimización de Hiperparámetros.

Índice

1. Objetivos

Objetivo General

El objetivo de esta monografía es realizar un estudio detallado del enfoque de Neural Architecture Search (NAS) basado en Aprendizaje por Refuerzo propuesto por Zoph y Le (2016). En particular, busco comprender en profundidad el método presentado en el trabajo original y reproducir, en la medida de lo posible, los componentes centrales del estudio utilizando PyTorch como framework de desarrollo.

Objetivos Específicos

Para dar cumplimiento al objetivo general, se plantean las siguientes metas particulares:

- Analizar teóricamente el procedimiento de búsqueda automática de arquitecturas convolucionales mediante controladores recurrentes y aprendizaje por refuerzo.
- Implementar y entrenar la arquitectura convolucional óptima de 15 capas (NASCNN15) identificada en el estudio original.
- Reproducir, de manera acotada, el mecanismo de búsqueda para validar el comportamiento del controlador en un espacio de búsqueda reducido.
- Evaluar el rendimiento de dicha arquitectura en el dataset CIFAR-10 y contrastar los resultados obtenidos frente a lo reportado en la bibliografía de referencia.

2. Alcance

En este trabajo delimito el estudio al procedimiento de búsqueda de arquitecturas convolucionales presentado por Zoph y Le (2016). En particular, me concentro en los mecanismos centrales del método: el controlador basado en aprendizaje por refuerzo, el espacio de acciones definido para construir arquitecturas y la evaluación de los modelos generados.

El alcance de la monografía excluye dos componentes del paper original. Por un lado, no abordo la búsqueda de arquitecturas recurrentes para tareas de regresión, dado que su implementación y análisis requieren un tratamiento específico que se aparta del foco principal de este estudio. Por otro lado, tampoco exploro el espacio completo de arquitecturas convolucionales propuesto por los autores. En su lugar, investigo un subconjunto acotado, centrado en configuraciones que permitan encontrar una mejora en el rendimiento en el espacio espacial de arquitecturas. Esta restricción permite mantener un enfoque realista y reproducible dentro de los límites temporales y computacionales disponibles.

3. Motivación

La motivación de esta monografía surge del interés en comprender y aplicar técnicas avanzadas de búsqueda automática de arquitecturas, un área que ha demostrado resultados competitivos frente a diseños manuales en múltiples dominios. El trabajo de Zoph y Le (2016) constituye un punto de referencia en la intersección entre aprendizaje profundo y aprendizaje por refuerzo, y representa un ejemplo claro de cómo un enfoque basado en agentes puede descubrir arquitecturas eficientes sin intervención humana directa.

Mi objetivo es profundizar en los aspectos conceptuales y prácticos del método, analizando tanto su formulación como sus implicaciones en el diseño moderno de redes neuronales. La posibilidad de reproducir, aunque sea parcialmente, los resultados del paper resulta especialmente valiosa, ya que permite conectar la teoría con la implementación real y evaluar los desafíos asociados a este tipo de búsquedas.

4. Introducción

En la última década, el aprendizaje profundo ha impulsado avances significativos en tareas complejas de visión por computadora y procesamiento del lenguaje natural. En visión, las redes neuronales convolucionales han alcanzado niveles de rendimiento sin precedentes en clasificación de imágenes (Krizhevsky, Sutskever e Hinton, 2012; He et al., 2016; Huang et al., 2017), mientras que en lenguaje los modelos recurrentes han mejorado sustancialmente el tratamiento de secuencias (Pascanu et al., 2013; Jozefowicz, Zaremba y Sutskever, 2015). Estos logros reflejan un cambio de paradigma: el diseño arquitectural ha pasado a ser un componente decisivo en el desempeño de los modelos.

Sin embargo, la construcción de arquitecturas profundas efectivas continúa requiriendo un alto grado de conocimiento experto y una gran cantidad de experimentación manual. Esta necesidad ha dado origen a la Búsqueda de Arquitecturas Neuronales (NAS), cuyo objetivo es automatizar el diseño de modelos mediante la exploración sistemática de espacios de arquitecturas.

Dentro de las estrategias propuestas, el enfoque basado en aprendizaje por refuerzo ha adquirido particular relevancia. Zoph y Le (2016) introdujeron un método en el cual un controlador recurrente genera descripciones de arquitecturas que luego se entrenan y evalúan, utilizando gradiente de política como regla de actualización (Williams, 1992). Este esquema permite que el agente aprenda, de forma iterativa, a proponer arquitecturas cada vez más competitivas sin intervención humana directa.

En esta monografía estudio en detalle el método de NAS con aprendizaje por refuerzo presentado por Zoph y Le, con el fin de comprender su funcionamiento y reproducir parcialmente sus resultados mediante una implementación en PyTorch. El enfoque se limita a la búsqueda convolucional, excluyendo explícitamente la parte del paper dedicada a arquitecturas recurrentes.

5. Estado del Arte

La búsqueda automática de arquitecturas neuronales ha sido abordada mediante diversos enfoques. En primer lugar, los métodos basados en optimización bayesiana han permitido explorar configuraciones dentro de arquitecturas predefinidas (Bergstra, Yamins y Cox, 2013). Si bien facilitan el ajuste de hiperparámetros como profundidad, número de filtros o tasas de aprendizaje, estos métodos no generan arquitecturas totalmente nuevas.

Otra línea de trabajo proviene de los algoritmos evolutivos, que modifican y seleccionan arquitecturas mediante operadores inspirados en procesos biológicos (Stanley y Miikkulainen, 2002). Estos métodos han demostrado capacidad para generar diseños competitivos, aunque suelen implicar elevados costos computacionales debido a la evaluación de numerosas arquitecturas candidatas. También se ha investigado el uso de "tejidos convolucionales" que representan espacios de diseño más flexibles (Saxena y Verbeek, 2016).

En el campo del aprendizaje por refuerzo, Baker et al. introdujeron MetaQNN, donde un agente basado en Q-learning selecciona secuencialmente el tipo de capa, tamaño de kernel, operaciones de *pooling* y demás atributos estructurales, utilizando la precisión de validación como señal de recompensa (Baker et al., 2016). Posteriormente, Pham et al. propusieron ENAS, que mantiene la idea de un controlador entrenado por gradiente de política, pero reduce drásticamente el costo computacional mediante el uso de pesos compartidos entre arquitecturas candidatas (Pham et al., 2018).

El trabajo de Zoph y Le (2016) se destaca como un punto de inflexión en esta línea. Su propuesta utiliza un controlador recurrente que produce códigos de arquitectura para redes convolucionales, los cuales son evaluados para obtener la recompensa que guía la actualización de la política. Este enfoque permitió descubrir arquitecturas convolucionales que rivalizan con diseños manuales de vanguardia, demostrando el potencial del aprendizaje por refuerzo para automatizar de forma efectiva la construcción de modelos profundos.

6. Marco Teórico

La formulación de la búsqueda de arquitecturas como un problema de optimización puede interpretarse dentro del marco del aprendizaje por refuerzo. En este contexto, un agente (el controlador) interactúa con un entorno donde cada acción corresponde a una decisión de diseño arquitectural —por ejemplo, seleccionar el tamaño de kernel o el número de filtros de una capa convolucional. Una secuencia completa de acciones genera una arquitectura candidata que será entrenada y evaluada.

La recompensa del agente se define a partir del rendimiento obtenido por la arquitectura generada, usualmente la precisión de validación. El objetivo del agente es maximizar la recompensa esperada, para lo cual se emplean métodos de gradiente de política. La regla REINFORCE (Williams, 1992) permite actualizar los parámetros de la política del controlador utilizando estimaciones de gradiente basadas en muestras de arquitecturas y sus recompensas asociadas.

En la propuesta de Zoph y Le, el agente está implementado mediante una red recurrente que codifica decisiones arquitecturales paso a paso. El espacio de búsqueda incluye configuraciones de capas convolucionales definidas por hiperparámetros como tipo de operación, tamaño del kernel y número de filtros. La arquitectura resultante de la secuencia generada por el controlador se entrena desde cero y su rendimiento sirve como señal de retroalimentación para actualizar la política.

Este marco permite formalizar la construcción de redes neuronales como un proceso de decisión secuencial, donde el aprendizaje por refuerzo guía la exploración del espacio de arquitecturas de forma dirigida y con potencial para descubrir estructuras eficaces.

7. Descripción del método de Zoph & Le (2016)

El método presentado por Zoph y Le (2016) constituye uno de los primeros enfoques efectivos de *Neural Architecture Search* (NAS) utilizando aprendizaje por refuerzo. Su idea central es entrenar un controlador recurrente capaz de generar arquitecturas neuronales mediante una secuencia de decisiones. Cada arquitectura generada se entrena y evalúa, y su rendimiento se utiliza como señal de recompensa para ajustar la política del controlador. De esta manera, el diseño arquitectural se convierte en un problema de decisión secuencial.

7.1. Formulación general del enfoque

La construcción de una arquitectura se modela como un proceso secuencial de longitud T. En cada paso t, el controlador emite una acción a_t extraída de un conjunto discreto de opciones estructurales:

$$a_t \sim \pi_\theta(a_t \mid s_t), \tag{1}$$

donde π_θ representa la política parametrizada por θ y s_t es el estado interno del controlador (la memoria de la red recurrente).

La secuencia de acciones

$$A = (a_1, a_2, \ldots, a_T) \tag{2}$$

codifica completamente una arquitectura convolucional. Una vez generada la arquitectura, esta se entrena desde cero y se evalúa sobre un conjunto de validación. El rendimiento obtenido produce una recompensa R, que se utiliza para actualizar la política mediante gradiente de política (REINFORCE) Figura 1.

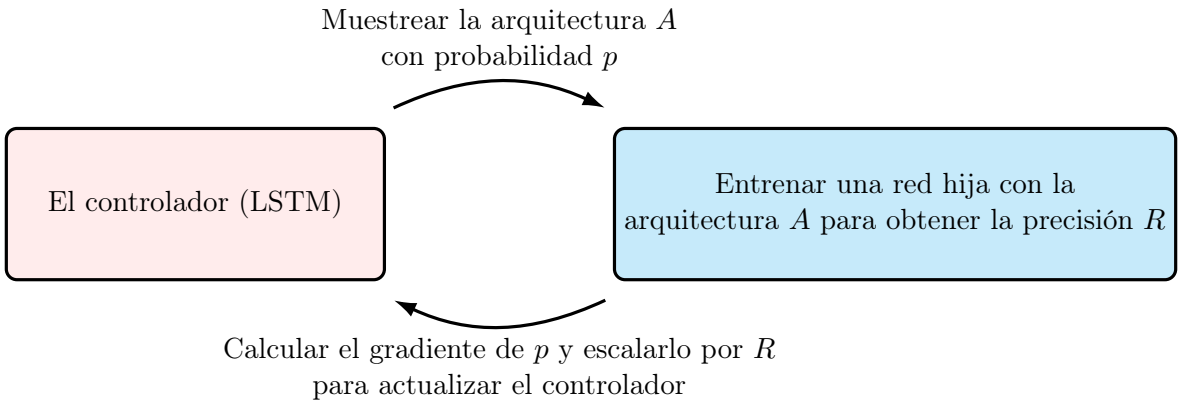

Figura 1: Ciclo principal del método NAS (traducción y adaptación).

7.2. Controlador basado en aprendizaje por refuerzo

El controlador se implementa como una red LSTM que genera decisiones arquitecturales paso a paso. El objetivo es maximizar la recompensa esperada:

$$J(\theta) = \mathbb{E}_{A\sim\pi_\theta}[R(A)]. \tag{3}$$

El gradiente de esta función objetivo se estima mediante la regla REINFORCE:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{A\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\,(R-b)\right], \tag{4}$$

donde R es la recompensa obtenida y b es un *baseline* que reduce la varianza del estimador.

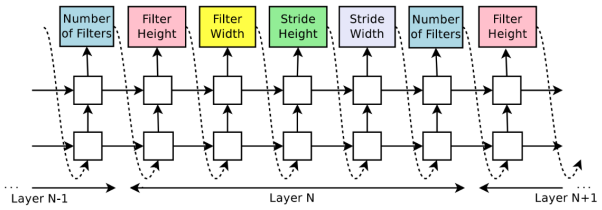

Figura 2: Ejemplo de una secuencia de acciones emitidas por el controlador LSTM.

7.3. Espacio de búsqueda convolucional

El espacio de búsqueda definido por Zoph y Le se compone de una serie de decisiones estructurales por capa, tales como:

- tipo de operación: convolución 3×3, 5×5, convolución separable, *max pooling*, *avg pooling*,
- número de filtros,
- tamaño del kernel,
- posibles conexiones de salto (*skip-connections*).

Para cada capa i, el controlador especifica:

$$a_i = (\text{op}_i, \text{kernel}_i, \text{filters}_i, \text{skip}_{i\to j}), \tag{5}$$

lo que permite construir redes profundas mediante decisiones discretas.

7.4. Entrenamiento y evaluación de arquitecturas

Una vez generada una arquitectura A, el procedimiento es el siguiente:

1. **Instanciación del modelo:** la secuencia de acciones se traduce en una CNN completa.
2. **Entrenamiento desde cero:** el modelo se entrena por un número fijo de épocas sobre el conjunto CIFAR-10.
3. **Evaluación:** se calcula la precisión en validación, que define la recompensa:

$$R = \text{Accuracy}_{\text{valid}}(A) \tag{6}$$

4. **Actualización del controlador:** se utiliza el gradiente REINFORCE para actualizar la política π_θ.

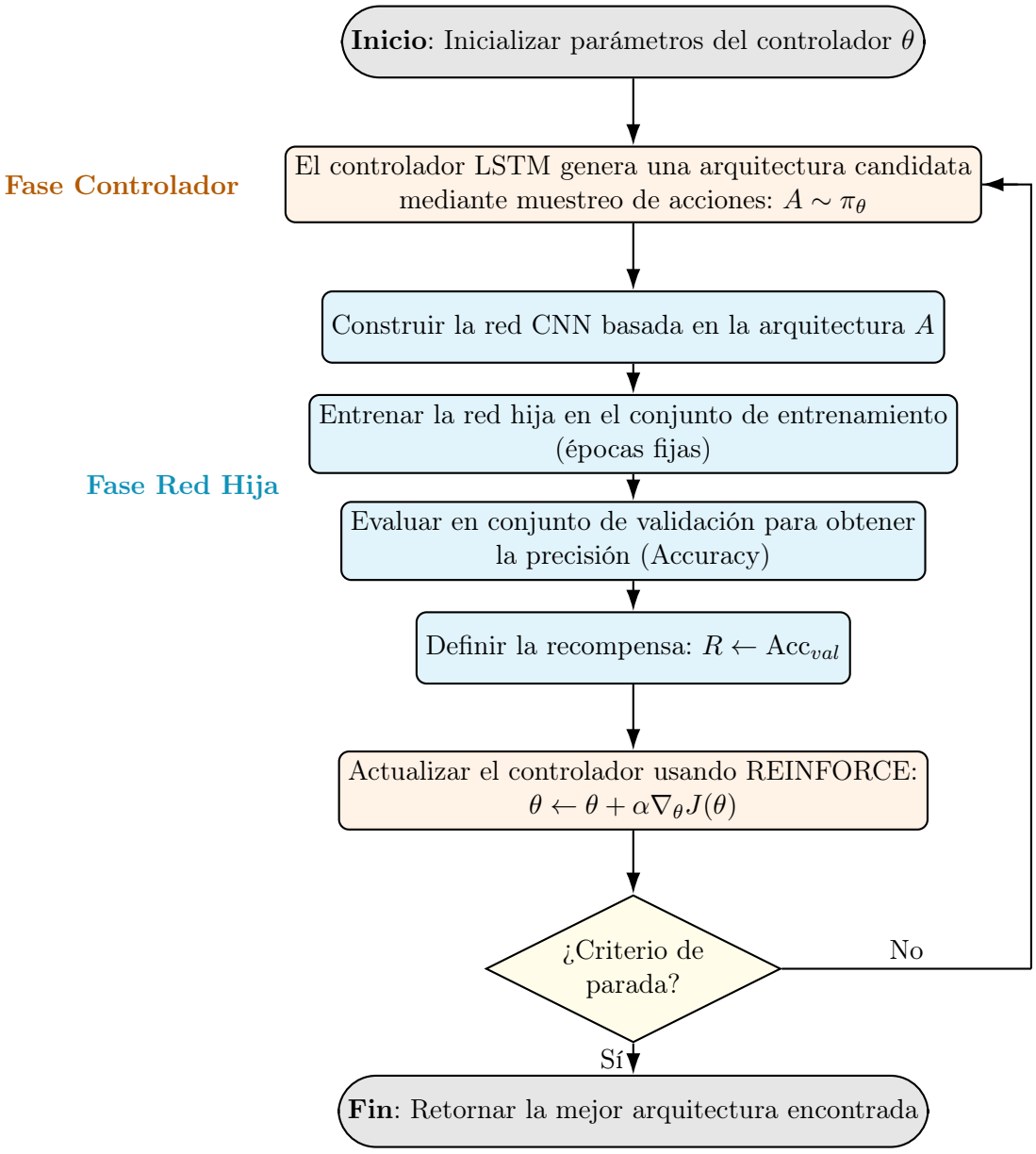

Figura 3: Diagrama de flujo del algoritmo NAS con Aprendizaje por Refuerzo.

7.5. Arquitectura convolucional final

Luego de un número suficiente de iteraciones del controlador, el método converge hacia arquitecturas de alto rendimiento. En el trabajo original, la arquitectura óptima encontrada es una CNN de 15 capas, con múltiples *skip-connections* y patrones repetidos descubiertos automáticamente.

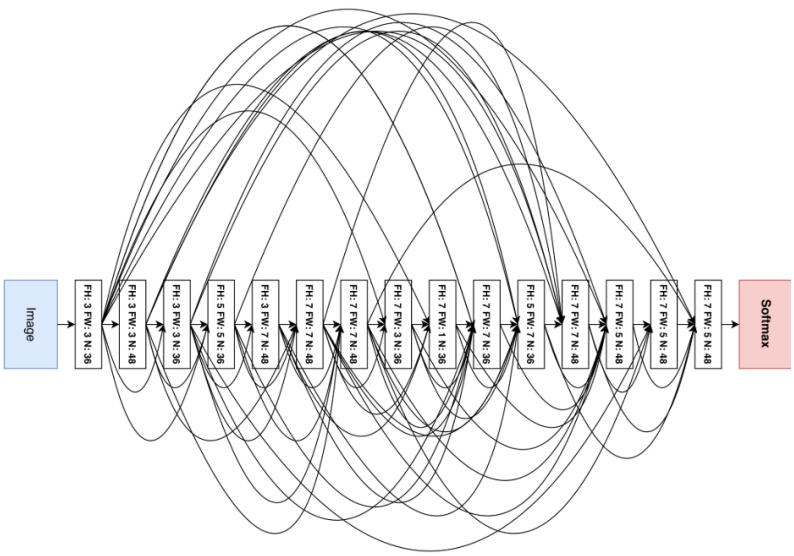

Figura 4: Arquitectura convolucional final descubierta por el proceso NAS. Imagen reportada en Anexo del paper.

8. Implementación en PyTorch

Este trabajo presenta una implementación completa del método de Búsqueda de Arquitecturas Neuronales con Aprendizaje por Refuerzo (NAS-RL) propuesto por Zoph & Le (Zoph y Le, 2016), adaptada para el dataset CIFAR-10 utilizando PyTorch 2.0+. La implementación se estructura como un sistema modular que integra generación de arquitecturas mediante redes LSTM, construcción dinámica de modelos CNN, y optimización vía gradiente de políticas (REINFORCE).

8.1. Estructura general del código

El proyecto se organiza en una arquitectura de componentes desacoplados que facilitan la experimentación del sistema NAS. La estructura principal se divide en dos subsistemas: el módulo de búsqueda de arquitecturas (`src/nas/`) y el pipeline de entrenamiento (`src/train_pipeline.py`), complementados por utilitarios de carga de datos y preprocesamiento.

8.1.1. Módulo NAS (`src/nas/`)

El núcleo del sistema de búsqueda se implementa en cinco componentes principales:

- **Controller** (`controller.py`): Red LSTM que actúa como generador de arquitecturas. Implementa un modelo recurrente de una capa con 100 unidades ocultas por defecto. El controlador genera secuencias de ADN arquitectónico: vectores de dimensión $N \times 4$, donde N representa el número de capas y cada cuádrupla codifica `[kernel_size, num_filters, stride, pool_size]`. El modelo contiene aproximadamente 11,000 parámetros entrenables en su configuración estándar.

- **Child Network Builder (`child_builder.py`)**: Constructor dinámico que transforma especificaciones de ADN en modelos CNN ejecutables. Implementa validación y recorte de componentes de ADN a rangos permitidos, insertando bloques de `Conv2d → BatchNorm2d → ReLU` según la especificación. Incluye lógica para manejo de arquitecturas inválidas mediante arquitecturas de respaldo y calcula padding automático para preservar resoluciones espaciales.
- **REINFORCE Optimizer (`reinforce.py`)**: Implementación del algoritmo REINFORCE (Williams, 1992) con baseline de promedio móvil exponencial (EMA). Configura el optimizador ADAM con tasa de aprendizaje inicial de 0.0006 (siguiendo el paper original), decaimiento exponencial cada 500 pasos ($\gamma = 0{,}96$), y regularización L2 con $\beta = 10^{-4}$. El baseline EMA utiliza un factor de suavizado $\alpha = 0{,}95$ para estabilizar las ventajas. Incluye recorte de gradientes (*gradient clipping*) con norma máxima de 1.0 para mitigar la inestabilidad del aprendizaje causada por actualizaciones excesivas.
- **NAS Trainer (`trainer.py`)**: Orquestador del proceso completo de búsqueda que coordina la interacción entre controlador, constructor de modelos, y optimizador. Gestiona el bucle de episodios, sampling de arquitecturas, entrenamiento de redes child, cálculo de recompensas, y actualización del controlador vía REINFORCE. Implementa un sistema de logging narrativo de 8 niveles jerárquicos para trazabilidad experimental, sistema de checkpoints con capacidad de reanudación, y seguimiento del mejor modelo encontrado. Incluye soporte opcional para *layer schedule* progresivo (incremento gradual de profundidad durante la búsqueda).
- **Utilities (`utils.py`)**: Funciones auxiliares para codificación/decodificación de ADN, validación de rangos, generación de ADN aleatorio, y visualización de arquitecturas. Centraliza los límites de componentes de ADN definidos en `configs.py`.

8.1.2. Pipeline de entrenamiento (`train_pipeline.py`)

El componente `TrainingPipeline` encapsula el proceso completo de entrenamiento de una arquitectura, ya sea generada por NAS o predefinida como NASCNN15. Implementa:

- Detección automática de dispositivo de cómputo (CUDA/MPS/CPU)
- Early stopping con paciencia configurable
- Scheduler de tasa de aprendizaje (`ReduceLROnPlateau`)
- Sistema de checkpoints con guardado del mejor modelo
- Cálculo y registro de métricas (loss, accuracy)
- Generación de visualizaciones (curvas de entrenamiento, matriz de confusión)
- Exportación de resultados y configuraciones experimentales en formato JSON

Este diseño modular permite reutilizar el pipeline tanto para entrenar las arquitecturas candidatas durante la búsqueda NAS como para el entrenamiento final de la arquitectura óptima encontrada.

8.1.3. Representación de ADN arquitectónico

Las arquitecturas se codifican mediante secuencias de ADN estructurado. Para una CNN de N capas, el ADN se representa como una matriz $\mathbf{D} \in \mathbb{Z}^{N\times 4}$, donde cada fila especifica:

$$\mathbf{D}_i = [\text{kernel}_i, \text{filters}_i, \text{stride}_i, \text{pool}_i] \tag{7}$$

Los rangos de valores permitidos se definen en `configs.py` siguiendo las restricciones del paper:

- **Kernel size**: $\{1, 3, 5, 7\}$ (dimensiones de filtro convolucional)
- **Num filters**: $\{24, 36, 48, 64\}$ (canales de salida)
- **Stride**: $\{1\}$ (fijo para CIFAR-10)
- **Pool size**: $\{1\}$ (sin pooling en experimento base)

Esta codificación permite al controlador explorar un espacio de búsqueda compacto pero expresivo, manteniendo la viabilidad computacional de cada arquitectura propuesta.

8.2. Adaptaciones respecto al paper original

Si bien la implementación sigue fielmente la metodología descrita en Zoph & Le (2016), se introdujeron adaptaciones pragmáticas dictadas por restricciones de recursos computacionales y consideraciones de viabilidad experimental:

8.2.1. Escala de la búsqueda

El paper original evalúa aproximadamente 12,800 arquitecturas sobre un cluster de 800 GPUs. Esta implementación proporciona configuraciones escalables:

- **Configuración "demo"**: 160 arquitecturas evaluadas (10 episodios $\times$ 16 children), diseñada para validación conceptual y pruebas de integración en hardware limitado.
- **Configuración "nasrlfull"**: 12,800 arquitecturas (1,280 episodios $\times$ 10 children), replicando la escala del paper con *layer schedule* progresivo.

8.2.2. Entrenamiento de arquitecturas candidatas (child networks)

El paper entrena cada arquitectura candidata por 50 épocas completas. Para la configuración "demo" se reduce a 20 épocas, manteniendo el resto de hiperparámetros (SGD con Nesterov momentum, $lr = 0{,}1$, $\beta = 0{,}9$, weight decay 10^{-4}). Esta reducción introduce un trade-off entre tiempo de búsqueda y precisión de la señal de recompensa, pero mantiene suficiente capacidad discriminatoria entre arquitecturas prometedoras y no prometedoras.

8.2.3. Función de recompensa

Siguiendo el paper, la recompensa R para una arquitectura se calcula como:

$$R = \left(\max_{k \in \text{last\_K}} \text{acc}_{\text{val}}^{(k)} \right)^3 \tag{8}$$

donde $K = 5$ (últimas 5 épocas) y el exponente cúbico enfatiza arquitecturas de alto rendimiento. Esta formulación estabiliza la señal de recompensa ante fluctuaciones estocásticas al final del entrenamiento y acentúa diferencias entre modelos competitivos.

8.2.4. Profundidad progresiva (layer schedule)

El paper implementa un esquema de profundidad incremental: comienza evaluando redes de 6 capas y aumenta progresivamente hasta alcanzar las 15 capas de NASCNN15. Esta implementación reproduce este mecanismo en la configuración "nasrlfull", incrementando la profundidad cada 1,600 arquitecturas evaluadas. En configuraciones reducidas ("demo"), el incremento ocurre cada 16 arquitecturas para preservar la dinámica del proceso dentro del presupuesto computacional reducido.

8.2.5. Simplificación del espacio de búsqueda

Para CIFAR-10, el paper fija stride=1 y elimina operaciones de pooling para preservar la resolución espacial de 32×32 píxeles. Esta implementación adopta la misma restricción, codificada en `DNA_LIMITS` (`configs.py`), simplificando el espacio de búsqueda y acelerando la convergencia del controlador.

8.2.6. Gestión de memoria y paralelización

A diferencia del entrenamiento paralelo de múltiples child networks en el paper, esta implementación evalúa arquitecturas secuencialmente para compatibilidad con hardware de consumidor (GPUs con 16 GB de VRAM).

8.3. Entrenamiento del controlador

El controlador LSTM se entrena mediante el algoritmo REINFORCE, una técnica de gradiente de políticas que maximiza la recompensa esperada $\mathbb{E}[R]$ ajustando los parámetros θ de la política π_θ.

8.3.1. Arquitectura del controlador

El controlador se implementa como una red LSTM de una capa:

- **Input size**: 1 (cada timestep consume un escalar del ADN anterior)
- **Hidden size**: 35 unidades (configuración "nasrlfull", según paper) o 100 (configuración "default")
- **Output**: Capa totalmente conectada que mapea el estado oculto final a un vector de dimensión $N \times 4$, representando el ADN generado
- **Inicialización**: Bias de la capa de salida inicializado a 0.01 (siguiendo el paper)

8.3.2. Procedimiento de sampling

Para generar una arquitectura candidata:

1. Se inicializa el contexto con una arquitectura base: $\mathbf{D}_0 = [10, 128, 1, 1]^\top \otimes \mathbf{1}_N$ (producto de Kronecker para replicar a N capas)
2. Se alimenta $\mathbf{D}_0$ a la LSTM como secuencia temporal
3. La capa de salida produce ADN crudo: $\mathbf{D'} = \text{FC}(\text{LSTM}(\mathbf{D}_0))$
4. Se escala y discretiza: $\mathbf{D} = \text{clip}(\lfloor |\mathbf{D'}| \times 100 \rfloor, 1, 512)$
5. Se valida y recorta $\mathbf{D}$ a rangos definidos en `DNA_LIMITS`

Este procedimiento asegura que todas las arquitecturas generadas sean sintácticamente válidas antes de construir el modelo correspondiente.

8.3.3. Actualización REINFORCE

Tras evaluar un batch de B arquitecturas $\{\mathbf{D}_i\}_{i=1}^{B}$ con recompensas $\{R_i\}_{i=1}^{B}$, se actualiza el controlador:

1. **Cálculo de ventaja (advantage)**:

$$A_i = R_i - b_t \tag{9}$$

   donde b_t es el baseline EMA:

$$b_t = \alpha \cdot b_{t-1} + (1-\alpha) \cdot \bar{R}_t, \quad \alpha = 0{,}95 \tag{10}$$

2. **Normalización de ventaja** (estabilización):

$$A'_i = \frac{A_i - \mu_A}{\sigma_A + \epsilon}, \quad \epsilon = 10^{-8} \tag{11}$$

3. **Loss de política** (REINFORCE con reconstrucción):

$$\mathcal{L}_{\text{policy}} = \frac{1}{B} \sum_{i=1}^{B} \left(-A'_i \cdot \|\mathbf{D}_i - \hat{\mathbf{D}}_i\|_2^2 \right) \tag{12}$$

   donde $\hat{\mathbf{D}}_i = \text{Controller}(\mathbf{D}_i)$ es la reconstrucción del ADN.

4. **Regularización L2**:

$$\mathcal{L}_{\text{reg}} = \beta \sum_{\theta} \theta^2, \quad \beta = 10^{-4} \tag{13}$$

5. **Loss total**:

$$\mathcal{L} = \mathcal{L}_{\text{policy}} + \mathcal{L}_{\text{reg}} \tag{14}$$

6. **Actualización de parámetros**:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L} \tag{15}$$

   con ADAM ($\eta_0 = 0{,}0006$) y gradient clipping ($\|\nabla\|_{\text{máx}} = 1{,}0$).

8.3.4. Hiperparámetros del controlador

Los hiperparámetros principales, siguiendo el paper y adaptados experimentalmente:

Parámetro	**Valor (paper)**	**Valor (implementación)**
Learning rate inicial (η_0)	0.0006	0.0006
Optimizador	ADAM	ADAM
Decay rate (γ)	0.96	0.96
Decay steps	-	500
L2 regularization (β)	10^{-4}	10^{-4}
Baseline EMA (α)	0.95	0.95
Gradient clipping	-	1.0
Hidden units (LSTM)	35	35

8.3.5. Convergencia y seguimiento

El sistema registra métricas por episodio para monitorear convergencia:

- **Mean reward**: $\bar{R}_{\text{ep}} = \frac{1}{B}\sum_{i=1}^{B} R_i$
- **Baseline EMA**: b_t
- **Mean advantage**: $\bar{A} = \frac{1}{B}\sum_{i=1}^{B} A'_i$
- **Learning rate**: η_t
- **Best reward global**: $\max_{i\in[1,t]} R_i$

Estos indicadores permiten detectar convergencia prematura, sobreajuste del controlador, o necesidad de ajustar hiperparámetros.

8.4. Entrenamiento de la arquitectura óptima

Tras finalizar la fase de Búsqueda de Arquitectura Neuronal (NAS) en un entorno de prueba, se procede al entrenamiento de la arquitectura que maximizó la función de recompensa. Este proyecto implementa la NASCNN15, una red compuesta por 15 capas convolucionales con conexiones densas (dense skip connections), tal como fue descubierta y definida en la investigación de referencia.

8.4.1. Arquitectura NASCNN15

NASCNN15 se estructura en 15 capas convolucionales organizadas con topología de grafo acíclico dirigido (DAG). Características principales:

- **Profundidad**: 15 capas convolucionales + capa de clasificación
- **Stride y pooling**: Todos los strides son 1, sin operaciones de max pooling (preserva resolución 32×32)
- **Número de filtros**: Alterna entre 36 y 48 canales según la capa
- **Tamaños de kernel**: Variados: 1×1, 3×3, 5×5, 7×7, 3×7, 7×3, 7×5, 5×7
- **Skip connections**: Concatenación densa por canales entre capas no adyacentes (hasta 9 capas predecesoras)
- **Normalización**: Batch Normalization después de cada convolución
- **Activación**: ReLU tras cada BatchNorm
- **Clasificador**: Global Average Pooling (GAP) + Fully Connected ($48 \to 10$ clases)
- **Parámetros totales**: $\sim$ 4,2 millones.

La topología exacta de conexiones salteadas se codifica en el método `forward()`, donde cada capa puede recibir la concatenación de hasta 9 mapas de características de capas anteriores, aumentando la capacidad expresiva sin incrementar drásticamente la profundidad. Figura 4

8.4.2. Configuración de entrenamiento

El entrenamiento de NASCNN15 sigue las mejores prácticas para CIFAR-10, alineándose con el paper:

- **Optimizador**: SGD con Nesterov momentum

$$v_{t+1} = \beta \cdot v_t + \nabla_\theta \mathcal{L}_t, \quad \theta_{t+1} = \theta_t - \eta_t(v_{t+1} + \beta \cdot \nabla_\theta \mathcal{L}_t) \tag{16}$$

  con $\beta = 0{,}9$ (momentum)

- **Learning rate**: Inicial $\eta_0 = 0{,}1$, con scheduler `ReduceLROnPlateau`
- **Weight decay**: $\lambda = 10^{-4}$
- **Función de pérdida**: `CrossEntropyLoss` sin label smoothing
- **Batch size**: 128
- **Épocas**: 300 con early stopping
- **Early stopping**: Detiene entrenamiento si la accuracy de validación no mejora por 20 épocas consecutivas.

8.4.3. Preprocesamiento de datos

Los datos de CIFAR-10 se preparan siguiendo protocolos estándar:

1. **Normalización**: Estandarización por canal RGB usando media y desviación estándar del conjunto de entrenamiento:

$$\mathbf{x}_{\text{norm}} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \quad \boldsymbol{\mu}, \boldsymbol{\sigma} \in \mathbb{R}^3 \tag{17}$$

2. **Augmentación (entrenamiento)**:
   - Random horizontal flip (p=0.5)
   - Random crop 32×32 con padding de 4 píxeles
3. **División**: 50,000 imágenes de entrenamiento (90 % train, 10 % validation) + 10,000 test

8.4.4. Pipeline de evaluación

Tras el entrenamiento, se evalúa NASCNN15 sobre el test set con las siguientes métricas:

- **Test accuracy**: Proporción de predicciones correctas
- **Top-5 accuracy**: Métrica complementaria (aunque menos relevante para 10 clases)
- **Curvas de entrenamiento**: Loss y accuracy en train/validation por época
- **Matriz de confusión**: Análisis de errores por clase

El sistema guarda automáticamente:

- Checkpoint del mejor modelo (`.pth`)
- Configuración completa del experimento (`.md`)
- Métricas de entrenamiento (`.json`)
- Visualizaciones (gráficos en PNG)

Nota metodológica: Todo el código fuente, configuraciones experimentales, y notebooks de análisis están disponibles en el repositorio del proyecto (`https://github.com/JoacoTschopp/NAS_LR_CNN`).

9. Resultados

9.1. Rendimiento del controlador

La ejecución del experimento de búsqueda de arquitectura neural (NAS) se completó exitosamente tras evaluar un total de 160 modelos candidatos a lo largo de 10 episodios. La Tabla 1 resume las métricas globales del proceso.

Tabla 1: Métricas de entrenamiento del controlador (Demo).

Métrica	**Valor**
Episodios totales	10
Arquitecturas evaluadas	160
Recompensa promedio inicial (R_0, Ep. 1)	0.4754
Recompensa promedio final (R_T, Ep. 10)	0.4856
Mejor recompensa global (Best Val Acc)	0.6046
Baseline EMA final (b_T)	0.4856
Learning rate del controlador (η)	0.0006
Tiempo total de búsqueda	$\sim$32.2 h

La Figura 5 muestra la evolución de la recompensa media. Se observa que, aunque el promedio por episodio se mantuvo estable alrededor de 0.48, el controlador fue capaz de explorar y alcanzar el máximo global en el Episodio 7.

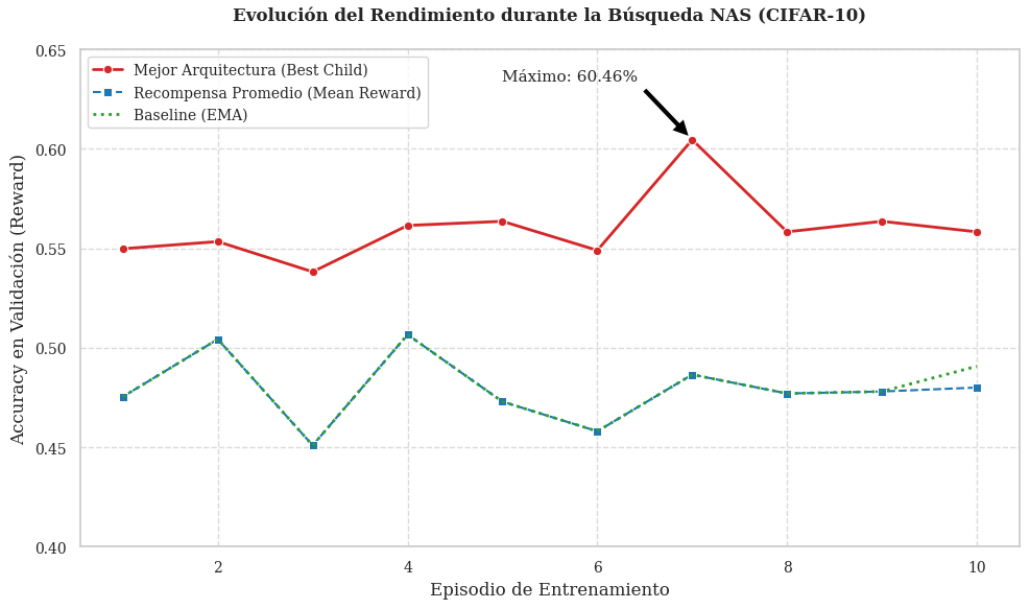

Figura 5: Evolución de la recompensa media por episodio y baseline EMA. El pico de desempeño se observó en el episodio 7.

9.2. Arquitectura encontrada: Análisis del ADN

El proceso de Búsqueda de Arquitectura Neuronal (NAS) culminó en el episodio 7 con el hallazgo de una configuración arquitectónica que maximizó la recompensa ($R \approx 0{,}6046$). Esta arquitectura no se define manualmente, sino que es el resultado de la decodificación de una secuencia de acciones discretas emitidas por el controlador LSTM.

A continuación, se presenta la representación cruda del "ADN" de la red, tal como fue generado por el controlador y registrado en el sistema.

9.2.1. Representación vectorial (ADN)

El ADN de la arquitectura es una lista de 13 vectores, donde cada vector de cuatro enteros $[k, f, s, p]$ codifica los hiperparámetros de una capa convolucional específica.

```
[
  [1, 2, 11, 1],    % Capa 1
  [1, 4, 8, 1],     % Capa 2
  [10, 2, 2, 8],    % Capa 3
  [2, 13, 13, 5],   % Capa 4
  [15, 4, 2, 11],   % Capa 5
  [24, 10, 7, 22],  % Capa 6
  [7, 13, 4, 25],   % Capa 7
  [4, 1, 3, 17],    % Capa 8
  [8, 10, 11, 17],  % Capa 9
  [11, 5, 9, 6],    % Capa 10
  [19, 1, 4, 5],    % Capa 11
  [17, 2, 3, 2],    % Capa 12
  [18, 12, 16, 11]  % Capa 13
]
```

9.2.2. Interpretación Estructural

La Tabla 2 detalla la arquitectura resultante tras decodificar el ADN.

Tabla 2: Decodificación del ADN de la mejor arquitectura (Episodio 7).

Capa	**Vector ADN**	**Kernel (K)**	**Filtros (F)**	**Stride (S)**	**Padding (P)**
1	`[1, 2, 11, 1]`	1×1	2	11	1×1
2	`[1, 4, 8, 1]`	1×1	4	8	1×1
3	`[10, 2, 2, 8]`	10×10	2	2	8×8
4	`[2, 13, 13, 5]`	2×2	13	13	5×5
5	`[15, 4, 2, 11]`	15×15	4	2	11×11
6	`[24, 10, 7, 22]`	24×24	10	7	22×22
7	`[7, 13, 4, 25]`	7×7	13	4	25×25
8	`[4, 1, 3, 17]`	4×4	1	3	17×17
9	`[8, 10, 11, 17]`	8×8	10	11	17×17
10	`[11, 5, 9, 6]`	11×11	5	9	6×6
11	`[19, 1, 4, 5]`	19×19	1	4	5×5
12	`[17, 2, 3, 2]`	17×17	2	3	2×2
13	`[18, 12, 16, 11]`	18×18	12	16	11×11

Análisis de la estructura descubierta: La arquitectura presenta patrones poco convencionales comparada con diseños humanos estándar:

1. **Stride y Padding:** El controlador ha seleccionado strides muy altos (ej. 11, 13, 16) que reducirían drásticamente la dimensión espacial de la imagen. Sin embargo, compensa esto con paddings masivos (ej. 22×22, 25×25).

2. **Kernels:** A diferencia de la tendencia moderna de usar kernels pequeños (3×3), esta red utiliza kernels muy grandes (15×15, 24×24).

3. **Filtros:** El número de filtros es extremadamente bajo (entre 1 y 13).

Tabla 3: Comparación: Arquitectura NAS Demo vs. Referencias.

Característica	**NAS (Este trabajo)**	**NASCNN15 (Paper)**
Número de capas	13	15
Parámetros totales	∼0.27 M	∼4.2 M
Kernels predominantes	Híbrido (1×1 a 24×24)	Estándar (3×3, 5×5)
Filtros promedio	∼6	36–48
Validation Accuracy	60.46 %	No reportado

9.3. Evaluación cuantitativa NASCNN15

La Tabla 4 resume el rendimiento de la arquitectura NASCNN15 replicada. Se reportan las métricas correspondientes a la mejor época de validación (época 67), modelo que fue utilizado para la evaluación final.

Tabla 4: Rendimiento de NASCNN15 en CIFAR-10: Reproducción vs. Original.

Métrica	**Esta reproducción**	**Paper original Zoph y Le, 2016**
Accuracy validación	91.30 %	–
Accuracy test	91.06 %	94.50 %
Loss final (validation)	0.3041	–
Parámetros	∼4.2 M	4.2 M
Épocas entrenadas	87 / 300	–
Early stopping	Sí	–

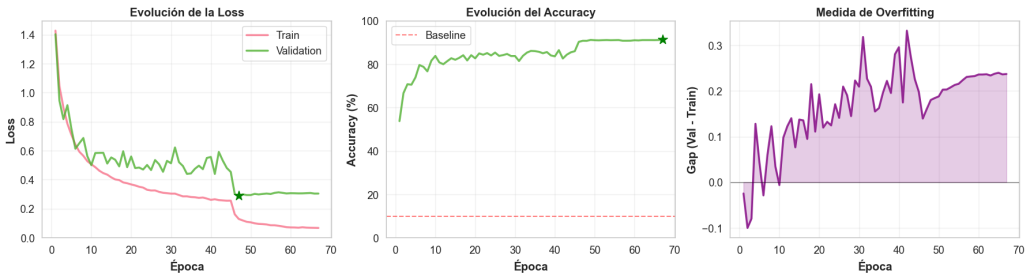

Figura 6: Curvas de entrenamiento de NASCNN15: loss y accuracy en train/validation por época.

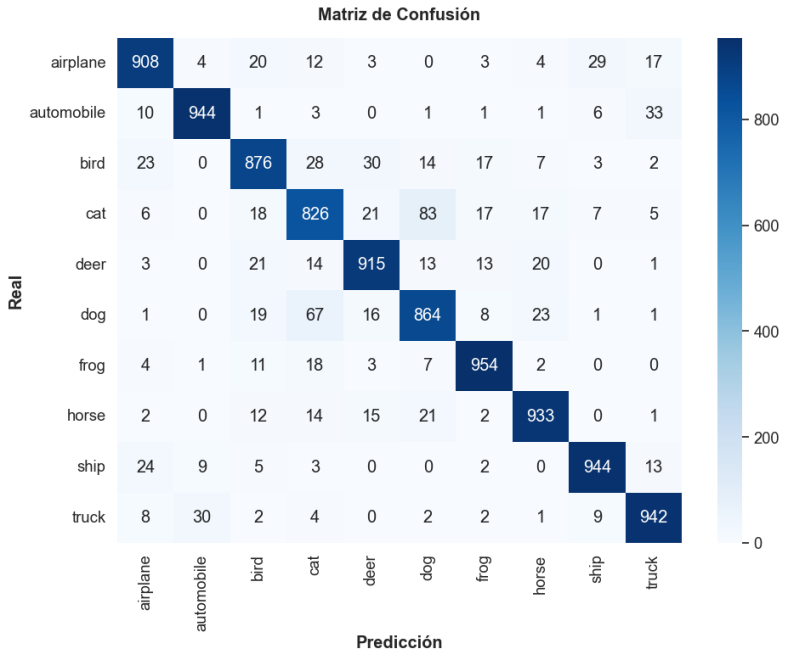

Figura 7: Matriz de confusión de NASCNN15 sobre el conjunto de test de CIFAR-10.

La Tabla 5 compara NASCNN15 con otras arquitecturas reportadas en la literatura. Cabe destacar que, aunque nuestra reproducción (91.06 %) no alcanza el 94.50 % del paper original, supera a arquitecturas clásicas como VGG y se acerca a los baselines de redes residuales tempranas, validando la eficacia del diseño encontrado por NAS.

Tabla 5: Comparación de NASCNN15 con arquitecturas de la literatura en CIFAR-10.

Modelo	**Parámetros**	**Test Acc.**	**Referencia**
Network in Network	–	91.19 %	Lin, Chen y Yan, 2014
ResNet-110	1.7 M	93.60 %	He et al., 2016
NAS v1 (Paper)	4.2 M	94.50 %	Zoph y Le, 2016
NASCNN15 (Reproducción)	4.2 M	91.06 %	–
DenseNet-BC (k=12)	0.8 M	94.80 %	Huang et al., 2017

10. Conclusiones

El desarrollo de este trabajo se centró en el estudio y la reproducción de la obra: *Neural Architecture Search with Reinforcement Learning* Zoph y Le, 2016. Este proceso reveló la notable complejidad intrínseca de la propuesta y, más significativamente, las dificultades que surgen ante la falta de detalles de implementación en la literatura científica de vanguardia. A lo largo de esta monografía, hemos abordado dos ejes principales: la mecánica de búsqueda mediante RL y la validación de la arquitectura reportada en el paper original.

Respecto a la implementación del algoritmo de búsqueda (NAS) mediante Aprendizaje por Refuerzo, se logró una ejecución exitosa de una prueba de concepto (“demo”). A pesar de las severas restricciones computacionales que limitaron la búsqueda a una fracción de los episodios reportados en el paper original, los resultados fueron sumamente positivos. El sistema fue capaz de evolucionar desde una inicialización aleatoria hasta encontrar una arquitectura no convencional que maximizó la recompensa en el entorno acotado. Esta experiencia práctica permitió validar el funcionamiento de la metodología NAS y comprender la dinámica entre el controlador (LSTM)

y la red hija, demostrando que, incluso en escenarios de recursos limitados, el algoritmo converge hacia soluciones optimizadas, aunque su despliegue en escenarios reales siga presentando barreras de entrada significativas.

En cuanto a la reproducción de la arquitectura de 15 capas (NASCNN15) propuesta como hallazgo en el paper, el principal desafío radicó en la escasez de especificaciones técnicas. La documentación original se limita a un diagrama de conexiones y menciona que los hiperparámetros fueron obtenidos mediante *grid search*, sin especificar los valores finales resultantes. Cabe destacar que, aunque los autores indicaron que la arquitectura estaría disponible en el repositorio oficial de TensorFlow, esta nunca fue publicada, lo que obligó a realizar una reconstrucción inferencial basada en la topología visual. A pesar de estas limitaciones y de contar con capacidades de búsqueda de hiperparámetros reducidas, la implementación alcanzó un *accuracy* en test del 91.06 %, un valor cercano al 94.5 % reportado, validando la solidez estructural del diseño descubierto.

Finalmente, es pertinente una reflexión crítica sobre la vigencia y el impacto de este trabajo. Si bien marcó un hito en 2016, la arquitectura encontrada cayó rápidamente en desuso, en parte debido a problemas de generalización. La aparición posterior de datasets de prueba robustos, como CIFAR-10.1 en 2018, evidenció que modelos optimizados excesivamente sobre un set de validación específico tienden a sobreajustarse; en nuestras pruebas internas, la red mostró dificultades para superar el 72 % en estos nuevos datos a pesar de mantener métricas altas en validación con el dataset de test original de CIFAR10.

En conclusión, la motivación y el valor perdurable del trabajo de Zoph y Le no reside tanto en la arquitectura específica encontrada, sino en la demostración empírica de que un algoritmo (NAS+RL) puede superar la intuición humana en el diseño de redes. Sin embargo, este logro viene acompañado de un costo computacional que era casi privativo en 2016 y que, aún hoy en la era del auge de los LLMs, sigue representando un desafío de infraestructura y costos.

Referencias

Baker, Bowen et al. (2016). «Designing neural network architectures using reinforcement learning». En: *arXiv preprint arXiv:1611.02167*.

Bergstra, James, Daniel Yamins y David Cox (jun. de 2013). «Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures». En: *Proceedings of the 30th International Conference on Machine Learning*. Ed. por Sanjoy Dasgupta y David McAllester. Proceedings of Machine Learning Research. Conference dates: 17–19 Jun. Atlanta, Georgia, USA: PMLR, págs. 115-123. URL: https://proceedings.mlr.press/v28/bergstra13.html.

He, Kaiming et al. (2016). «Deep residual learning for image recognition». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*, págs. 770-778.

Huang, Gao et al. (2017). «Densely connected convolutional networks». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*, págs. 4700-4708.

Jozefowicz, Rafal, Wojciech Zaremba e Ilya Sutskever (2015). «An empirical exploration of recurrent network architectures». En: *International conference on machine learning*. PMLR, págs. 2342-2350.

Krizhevsky, Alex, Ilya Sutskever y Geoffrey E Hinton (2012). «Imagenet classification with deep convolutional neural networks». En: *Advances in neural information processing systems* 25.

Lin, Min, Qiang Chen y Shuicheng Yan (2014). «Network in network». En: *International Conference on Learning Representations (ICLR)*.

Pascanu, Razvan et al. (2013). «How to construct deep recurrent neural networks». En: *arXiv preprint arXiv:1312.6026*.

Pham, Hieu et al. (2018). «Efficient neural architecture search via parameters sharing». En: *International conference on machine learning*. PMLR, págs. 4095-4104.

Saxena, Shreyas y Jakob Verbeek (2016). «Convolutional neural fabrics». En: *Advances in neural information processing systems* 29.

Stanley, Kenneth O y Risto Miikkulainen (2002). «Evolving neural networks through augmenting topologies». En: *Evolutionary computation* 10.2, págs. 99-127.

Williams, Ronald J. (1 de mayo de 1992). «Simple statistical gradient-following algorithms for connectionist reinforcement learning». En: *Machine Learning* 8.3, págs. 229-256. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

Zoph, Barret y Quoc V Le (2016). «Neural architecture search with reinforcement learning». En: *arXiv preprint arXiv:1611.01578*.