REINFORCEMENT LEARNING FOR WEIGHT INITIALIZATION

DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF THE DEGREE
OF

MASTER OF SCIENCE
IN
APPLIED MATHEMATICS

Submitted by

YAMAN (23/MSCMAT/48)

DEEPANSHU SHONDHANI (23/MSCMAT/69)

Under the supervision of

Prof. ANJANA GUPTA

DEPARTMENT OF APPLIED MATHEMATICS
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi 110042

MAY, 2025

DEPARTMENT OF APPLIED MATHEMATICS
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

We, **YAMAN** and **DEEPANSHU SHONDHANI**, Roll No's **- 23/MSCMAT/48** and **23/MSCMAT/69** students of M.Sc (**Applied Mathematics**) ,hereby declare that the Dissertation titled **'Reinforcement learning for weight initialization'** which is submitted by us to the Department of Applied Mathematics, Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of the Master of Science degree is original and not copied from any source without proper citation. The matter presented in the thesis has not been submitted by us for the award of any other degree of this or any other Institute.

Delhi | Yaman , Deepanshu Shondhani

Date: 26.05.2025 | 23/MSCMAT/48, 23/MSCMAT/69

This is to certify that the student has incorporated all the corrections suggested by the examiners in the dissertation and the statement made by the candidate is correct to the best of our knowledge.

Signature of Supervisor | **Signature of External Examiner**

DEPARTMENT OF APPLIED MATHEMATICS
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

<u>CERTIFICATE</u>

I hereby certify that the dissertation titled **"Reinforcement learning for weight initialization"** which is submitted by **Yaman** and **Deepanshu Shondhani**, Roll No's – **23/MSCMAT/48** and **23/MSCMAT/69**, Department of Applied Mathematics ,Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of the degree of Master of Science , is a record of the project work carried out by the students under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: Delhi **Prof. Anjana Gupta**

Date: 26.05.2025 **SUPERVISOR**

DEPARTMENT OF APPLIED MATHEMATICS
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

ACKNOWLEDGEMENT

We wish to express our sincerest gratitude to Prof Anjana Gupta for her continuous guidance and mentorship that she provided us during the project. She showed us the path to achieve our goals by explaining all the tasks to be done and explained to us the importance of this project as well as its industrial relevance. She was always ready to help us and clear our doubts regarding any hurdles in this project. Without her constant support and motivation, this dissertation would not have been successful.

Delhi — Yaman , Deepanshu Shondhani

Date: 26.05.2025 — 23/MSCMAT/48, 23/MSCMAT/69

Abstract

Effective neural network weight initialization is crucial for successful training, yet standard methods often rely on assumptions violated by modern architectures and advanced activation functions like Swish. This dissertation details a study investigating the feasibility of using Reinforcement Learning (RL) to tune a scaling factor for He initialization when employing Swish activations. An RL agent explored different scaling factors, evaluating them by training a small convolutional neural network on CIFAR-10 for a 7-epoch proxy task. Over 200 episodes, the RL agent demonstrated learning, converging towards a specific range of scaling factors that optimized the 7-epoch validation accuracy. This preliminary investigation highlights the functionality of the RL framework for initialization tuning and underscores the importance of evaluating the fidelity of short-term proxy tasks in predicting longer-term training performance, informing subsequent research into more complex symbolic initialization discovery.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 The Importance of Weight Initialization in Deep Learning

The initial values assigned to the weights within a deep neural network are not merely arbitrary starting points; they profoundly dictate the subsequent training dynamics and ultimately, the performance of the learned model. The process of deep learning optimization, typically achieved through gradient-based methods like stochastic gradient descent (SGD) and its variants, is highly sensitive to these initial conditions. An improperly initialized network can suffer from a myriad of issues that impede effective learning, making weight initialization a cornerstone of successful deep learning model development.

One of the most critical impacts of initial weights is on the convergence speed of the training process. When weights are too small, the gradients propagated back through the network during backpropagation can vanish, becoming infinitesimally small. This phenomenon, known as the vanishing gradient problem, effectively halts learning in earlier layers of deep networks, as updates to their weights become negligible [1]. Conversely, if initial weights are excessively large, gradients can explode, leading to numerical instability, oscillations during training, and divergence of the optimization process—the exploding gradient problem [1]. Both scenarios significantly prolong training times, or worse, prevent the network from converging to a meaningful solution at all.

Beyond convergence speed, the choice of initial weights directly influences the final performance of the model. Deep learning optimization landscapes are often complex and non-convex, characterized by numerous local minima and saddle points. A poor initialization can trap the optimization algorithm in a suboptimal local minimum, preventing the model from reaching a high-performing global or near-global minimum [2]. The proximity of the initial weight configuration to a favorable region in the loss landscape can dramatically affect the model's capacity to generalize well to unseen data. Seminal works by Glorot and Bengio [1] and He et al. [2] unequivocally established the critical role of thoughtful weight initialization in mitigating these challenges and enabling the training of deeper and more effective neural networks. Their research demonstrated that carefully scaled initializations could promote stable gradient flow, leading to faster convergence and improved final model performance.

1.2 Challenges with Standard Initialization Heuristics

The groundbreaking work of Glorot and Bengio [1] and He et al. [2] introduced data-dependent initialization schemes that dramatically improved the training of deep networks. Glorot and Bengio's "Xavier" initialization proposed scaling weights based on the number of input and output units of a layer, aiming to maintain variance of activations and gradients across layers. This approach was particularly effective for activation functions that are symmetric around zero, such as tanh and sigmoid, as it assumed a linear regime of operation for these activations around zero.

He et al. [2] further refined this concept, developing initialization strategies specifically tailored for Rectified Linear Unit (ReLU) activation functions and their variants. Recognizing that ReLUs rectify negative inputs to zero, breaking the symmetry assumption of Xavier, He initialization scaled weights by considering only the number of input units to a layer. This adjustment proved crucial for mitigating the vanishing gradient problem in networks employing ReLU, allowing for the training of significantly deeper architectures.

Despite their widespread success, both Glorot and He initialization heuristics rely on specific assumptions about the activation functions used within the network. These assumptions primarily include linearity or piecewise linearity and symmetry around zero (for Glorot). However, the landscape of activation functions in deep learning is continuously evolving, with novel functions emerging that do not conform to these traditional assumptions. A prime example is the Swish activation function, introduced by Ramachandran et al. [3], defined as $f(x) = x \cdot \sigma(\beta x)$, where $\sigma(x)$ is the sigmoid function and β is a learnable or fixed parameter.

Swish exhibits several desirable properties, including being smooth and non-monotonic. Its non-monotonicity, where the output sometimes decreases even as the input increases, is a key departure from traditional activations. This characteristic, along with its reliance on the sigmoid function, violates the linear or symmetric assumptions underpinning both Glorot and He initialization schemes. Consequently, analytically deriving optimal variance scaling factors for Swish becomes significantly more challenging, if not intractable, using the methods employed by Glorot and He. This limitation highlights a critical gap: as novel activation functions are developed, the reliance on manual, analytical derivation of initialization parameters becomes unsustainable and potentially suboptimal.

1.3 Reinforcement Learning for Automated Discovery

The challenges associated with manually deriving optimal initialization parameters for increasingly complex and non-standard activation functions underscore the need for automated discovery mechanisms. Reinforcement Learning (RL) presents a powerful paradigm for addressing such problems. RL involves an agent learning to make a sequence of decisions by interacting with an environment to maximize a cumulative reward signal. The agent, through trial and error, explores the environment, performs actions, observes the consequences, and adjusts its policy to achieve its objectives.

RL has demonstrated remarkable success in a wide array of automated discovery tasks, particularly in the realm of machine learning itself. A notable example is its application in Neural Architecture Search (NAS) [4], where RL agents are trained to design optimal neural network architectures for specific tasks. This success illustrates RL's capability to navigate vast search spaces and learn complex relationships between design choices and performance outcomes.

Given its proven ability to learn optimal strategies in complex, high-dimensional spaces, we propose RL as a promising approach for finding or tuning initialization parameters and even discovering novel initialization formulas. Instead of relying on analytical derivations based on simplifying assumptions, an RL agent can learn to select initialization parameters that empirically lead to better training dynamics and model performance, effectively adapting to the nuances of specific activation functions and network architectures. This data-driven approach could overcome the limitations of traditional heuristic-based methods, paving the way for more robust and generalizable initialization strategies.

1.4 Research Objectives

Building upon the insights from the pilot investigation and the identified challenges, this dissertation aims to explore the potential of Reinforcement Learning for automated weight initialization tuning. Specifically, we will address the following research questions:

- **Q1:** Can an RL framework effectively learn to tune a continuous scaling parameter for a standard weight initialization scheme (He) when used with a non-standard activation function (Swish)?
- **Q2:** What are the learning dynamics of the RL agent, and what characteristics does the reward landscape exhibit in this simplified initialization tuning problem?
- **Q3:** How well does a short-term proxy task (e.g., 7-epoch training) predict the utility of an identified initialization parameter for longer, more extensive training durations?
- **Q4:** What are the implications of these findings for designing more complex RL-based searches for novel, potentially symbolic, initialization formulas?

1.5 Scope and Contributions of this Dissertation

Scope: This dissertation focuses on a pilot investigation into the application of Reinforcement Learning for weight initialization. Specifically, we will concentrate on tuning a single continuous scaling factor 's' for the existing He initialization scheme, but in conjunction with the non-standard Swish activation function. The experiments will be conducted on a specific Convolutional Neural Network (CNN) architecture trained on the CIFAR-10 dataset.

Contributions:

- Demonstration of a functional RL framework for tuning an initialization parameter.
- Empirical evidence of RL agent learning and convergence in this task.
- Analysis of proxy task fidelity for initialization tuning.
- Insights to guide future, more ambitious research in automated initialization discovery.

Chapter 2

LITERATURE REVIEW

2.1. Deep Neural Networks

Deep neural networks (DNNs) have revolutionized machine learning, achieving unprecedented performance in computer vision, natural language processing, and reinforcement learning. This section explores their fundamental concepts and architectures.

2.1.1. Basic Concepts: Neurons, Layers, Weights, and Biases

Neural networks are computational models inspired by biological brains [5]. The artificial neuron is the basic building block, processing signals from connections that have adjustable weights [5]. Neurons are organized into an input layer, one or more hidden layers for computations, and an output layer. A network with multiple hidden layers is a deep neural network [5].

Each neuron computes a weighted sum of inputs plus a bias term, followed by an activation function (f) that introduces non-linearity. The pre-activation value for a neuron in layer l is:

$$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}$$

Where $W^{(l)}$ is the weight matrix, $a^{(l-1)}$ is the previous layer's output, and $b^{(l)}$ is the bias vector. The output is then $a^{(l)} = f(z^{(l)})$ [6]. This process occurs during forward propagation.

2.1.2. Forward and Backward Propagation

Forward propagation calculates and stores intermediate variables from the input to the output layer [6]. Input data traverses layer by layer, computing outputs until the final layer produces a result used to calculate loss.

Backward propagation is the core algorithm for training DNNs. It calculates the gradient of the loss function with respect to each weight, enabling updates that minimize loss [7]. The error signal propagates backward, efficiently computing gradients using the chain rule [6].

For a weight $w_{ij}^{(l)}$:

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial C}{\partial w_{ij}^{(l)}} = -\eta \delta_i^{(l)} \cdot a_j^{(l-1)}$$

Where η is the learning rate, C is the cost function, $\delta_i^{(l)}$ is the error signal, and $a_j^{(l-1)}$ is the activation from the previous layer [8]. The bias update is $\Delta b_i^{(l)} = -\eta \delta_i^{(l)}$. This recursive process enables efficient training of deep networks [6].

2.1.3. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are specialized DNNs for grid-like data like images, excelling in tasks such as image classification (e.g., CIFAR-10). CNNs typically comprise three main layer types [5]:

- **Convolutional layers:** Apply learnable filters (kernels) to detect local patterns like edges and textures, generating feature maps.
- **Pooling layers:** Reduce spatial dimensions (e.g., max pooling), achieving translation invariance and reducing computational complexity.
- **Fully connected layers:** At the end, these integrate information for final predictions, similar to traditional neural networks.

CNNs' hierarchical structure mirrors the visual cortex, allowing them to learn increasingly abstract representations, making them effective for multi-level pattern recognition tasks like CIFAR-10 [5].

2.2. Activation Functions

Activation functions are crucial in neural networks, introducing the non-linearity necessary for learning complex patterns. Without them, deep networks would simply behave as linear regression models, regardless of their depth. They transform the linear combination of inputs and weights into non-linear outputs, allowing neural networks to approximate arbitrary functions and learn hierarchical representations [5]. They also act as gates, controlling information flow by determining neuron activation intensity.

2.2.1. Traditional Activations: Sigmoid and Tanh

Early neural networks commonly used sigmoid and hyperbolic tangent (tanh) functions.

- **Sigmoid Function:** $\sigma(x) = \frac{1}{1+e^{-x}}$, maps inputs to $[0, 1]$.
- **Tanh Function:** $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, maps inputs to $[-1, 1]$.

Both functions suffer from saturation (gradients approach zero for large/small inputs) and the vanishing gradient problem, where gradients diminish exponentially across layers during backpropagation, making deep network training difficult [9]. They also involve computationally expensive exponential operations. These limitations spurred the development of more efficient alternatives.

2.2.2. ReLU and its Variants

The Rectified Linear Unit (ReLU) and its variants have largely supplanted traditional activations in modern deep learning. ReLU is defined as $f(x) = \max(0, x)$. Its advantages include computational efficiency, sparse activation, and no saturation for positive inputs, mitigating vanishing gradients [10]. However, ReLU can suffer from the "dying ReLU" problem, where neurons become permanently inactive.

To address this, several variants emerged:

- **Leaky ReLU:** $f(x) = \max(\alpha x, x)$ ($\alpha \approx 0.01$), allows a small positive gradient for negative inputs, preventing dying ReLUs [11].
- **Parametric ReLU (PReLU):** Similar to Leaky ReLU, but α is learned during training [12].
- **Exponential Linear Unit (ELU):** $f(x) = x$ for $x > 0$ and $f(x) = \alpha(e^x - 1)$ for $x \leq 0$. ELU combines ReLU's benefits with negative values that push mean activations closer to zero, potentially improving learning dynamics [13].

2.2.3. Swish/SiLU

Swish, also known as Sigmoid Linear Unit (SiLU), is a recent activation defined as $f(x) = x \cdot \sigma(\beta x)$, where σ is the sigmoid function and β is a trainable parameter or fixed constant [14]. When $\beta = 1$, it simplifies to SiLU: $f(x) = x \cdot \sigma(x)$.

Swish's effectiveness comes from its properties:

- **Smoothness:** It's smooth everywhere, potentially aiding optimization [14].
- **Non-monotonicity:** It decreases slightly for negative values before increasing, allowing for more complex function modeling [14].
- **Self-gating mechanism:** The sigmoid component dynamically controls the linear component, providing a form of attention [14].
- **Bounded below and unbounded above:** It combines the boundedness of sigmoid for negative inputs with ReLU's non-saturating behavior for positive inputs.

Empirically, Swish often outperforms ReLU in deep networks, leading to improved accuracy and faster convergence, especially in architectures with 40+ layers [14]. However, its non-standard behavior, particularly its non-monotonicity and negative outputs, means that traditional initialization strategies—which make assumptions about activation function behavior—may not be optimal, representing a significant gap in current literature.

2.3 Weight Initialization Strategies

Weight initialization is critical for deep neural networks, impacting convergence speed and stability. Proper initialization aims to mitigate the vanishing and exploding gradient problems, fundamental challenges in deep learning.

2.3.1. The Vanishing and Exploding Gradient Problem

Vanishing gradients occur when gradients diminish significantly as they propagate backward through deep networks, hindering learning in early layers. This is often exacerbated by traditional activations like sigmoid/tanh [9]. Conversely, exploding gradients arise when gradients become excessively large, leading to unstable training and numerical overflow [15]. Both issues stem from the multiplicative nature of gradient propagation across layers, where magnitudes of weights and activation derivatives can cause exponential decay or growth. Effective initialization ensures stable variance of activations and gradients throughout the network.

2.3.2. Weight Initialization Methods

Various strategies have been developed to initialize neural network weights, each with different theoretical bases and applicability. These methods are summarized below:

Method	Key Idea	Applicable Activations	Formula (if any)	Notes
Random Init.	Small random values	Any (not optimal)	$W \sim U[-r, r]$ or $N(0, \sigma^2)$	Simplest; no variance guarantees; inconsistent performance.
Nguyen-Widrow	Efficient use of active region	Sigmoid (shallow nets)	Scaled by neuron count	Empirical; limited to shallow networks; improves convergence in simple cases.
Glorot/Xavier [9]	Preserve variance of activations & gradients across layers	Symmetric (tanh, sigmoid, linear)	$\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right]$	Balances forward/backward variance; assumes zero mean and linear-around-zero activations.
He [12]	Preserve variance for ReLU-type activations	ReLU and variants	$W \sim N\left(0, \frac{2}{\text{fan\_in}}\right)$	Compensates for ReLU's variance reduction; effective for deep ReLU networks.
LSUV [16]	Ensure unit variance for each layer's output empirically	Any	Data-dependent: orthonormal init, then normalize with forward pass.	Data-dependent; adaptive to architecture/data; often provides better initial conditions.
Orthogonal [17]	Preserve norms during forward propagation	Any (esp. very deep nets, RNNs)	Initialize as random orthogonal matrices.	Helps maintain stable gradient flow; extended to convolutional layers.

2.4 RL for Hyperparameter Optimization and AutoML

Hyperparameter optimization (HPO) is a critical aspect of deep learning, significantly impacting model performance. This section explores how reinforcement learning (RL) has been applied to automate hyperparameter selection, a key component of Automated Machine Learning (AutoML).

2.4.1. Overview of RL for Hyperparameter Optimization

Traditional HPO methods, such as grid search, random search, or Bayesian optimization, often struggle with the high-dimensional, non-differentiable nature of hyperparameter spaces and the sequential decision-making involved in tuning [18]. Reinforcement learning offers a promising alternative by framing hyperparameter optimization as a sequential decision process where:

- **States:** Represent the current model configuration and its performance.
- **Actions:** Involve making specific hyperparameter adjustments.
- **Rewards:** Are derived from improvements in validation metrics.
- **Policy:** Is the strategy for selecting hyperparameters based on the current state.

This formulation leverages RL's strengths in sequential decision-making under uncertainty, allowing for adaptive exploration of the hyperparameter space based on feedback from previous trials [18]. Recent research indicates that RL-based approaches can surpass traditional methods in both final performance and computational efficiency, especially for complex hyperparameter landscapes [18].

2.4.2. Neural Architecture Search (NAS)

Neural Architecture Search (NAS) stands as one of the most successful applications of RL to AutoML, focusing on automating the design of neural network architectures. In pioneering work by Zoph and Le (2017), a recurrent neural network (RNN) controller was trained with reinforcement learning to generate neural network architectures [19]. The controller outputs descriptions of neural networks, which are then trained to completion. The validation accuracy of these trained networks serves as the reward signal for updating the controller's policy.

Mathematically, the NAS framework aims to maximize:

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R]$$

Where θ_c represents the controller parameters, $a_{1:T}$ is a sequence of architecture decisions, and R is the validation accuracy of the generated architecture [19]. This approach has successfully produced architectures that rival or even surpass the best human-designed networks on tasks like image classification and language modeling. For instance, on CIFAR-10, RL-based NAS achieved a test error rate of 3.65%, outperforming previous state-of-the-art models [19].

2.4.3. Positioning Current Research

While Neural Architecture Search has garnered significant attention, applying RL principles to optimize specific hyperparameters within established architectures remains an important and complementary approach. The current research is positioned in this area, specifically focusing on the critical hyperparameter of weight initialization scaling factors, particularly for modern activation functions like Swish/SiLU.

This focused approach offers several advantages:

- **Targeted Optimization:** By concentrating on a specific hyperparameter with theoretical significance, the method can leverage domain knowledge while benefiting from RL's sequential decision-making capabilities.
- **Computational Efficiency:** Optimizing initialization factors requires significantly fewer computational resources compared to a full architecture search, potentially leading to substantial performance improvements with less overhead.
- **Generalizability:** Findings regarding optimal initialization strategies are often generalizable across multiple architectures and tasks, unlike NAS which typically produces task-specific architectures.

The current research applies reinforcement learning principles to fine-tune initialization scaling factors, directly addressing a recognized gap in the literature concerning optimal initialization strategies for modern activation functions [14]. This represents a novel application of RL within the broader AutoML context, concentrating on a critical hyperparameter that profoundly influences neural network training dynamics and final performance.

Chapter 3

METHODOLOGY

This chapter details the Reinforcement Learning (RL) framework developed and employed to investigate the automated tuning of a scaling factor for He weight initialization in a Convolutional Neural Network (CNN) utilizing Swish activation functions. The methodology encompasses the design of the child neural network, the parameterized initialization scheme, the RL environment, and the RL agent.

3.1 Overview of the RL Framework

The core of this research involved an RL agent interacting with a custom-designed environment. In each interaction step (episode), the RL agent proposed a continuous value representing a scaling factor, s. This factor was then used to initialize the weights of a "child" CNN. The child network was subsequently trained on the CIFAR-10 dataset for a fixed, short number of epochs (the "proxy task"). The validation accuracy achieved by the child network on this proxy task served as a reward signal, guiding the RL agent's learning process. The objective of the RL agent was to learn a policy that selected scaling factors leading to higher rewards, thereby identifying s values optimal for the proxy task's performance.

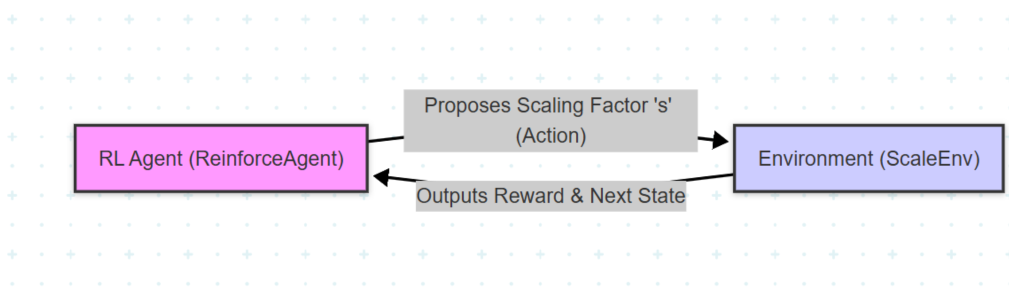

Figure 3.1: RL Framework for Initialization Scaling Factor Tuning. A block diagram showing: [RL Agent] → action (scaling factor s) → [Environment (ScaleEnv)] → trains [Child CNN] → returns reward (validation accuracy) & next state → [RL Agent].

3.2 Child Neural Network Architecture (Child CNN)

A relatively small CNN was designed as the "child network" for evaluation within the RL loop. The architecture was chosen to be sufficiently complex to represent common image classification tasks while remaining computationally inexpensive for rapid training and evaluation across numerous RL episodes.

The child network, `SimpleCNN`, is a convolutional neural network designed for image classification. It processes 3-channel input images (e.g., CIFAR-10 images of size $32 \times 32 \times 3$). The architecture is detailed below:

- **Input:** 3-channel images (e.g., CIFAR-10 images of size $32 \times 32 \times 3$).
- **Convolutional Block 1 (`self.conv1`):** Consists of a convolutional layer followed by SiLU activation and max pooling.
- **Convolutional Block 2 (`self.conv2`):** Similar to Block 1, but with increased feature maps.
- **Convolutional Block 3 (`self.conv3`):** Similar to Block 2, with further increased feature maps.
- **Flattening:** The output of the final pooling layer is flattened from $128 \times 4 \times 4$ to 2048 features.
- **Fully Connected Layer 1 (`self.fc1`):** A dense layer with 2048 input features and 256 output units, followed by SiLU activation.
- **Output Layer (`self.fc2`):** A fully connected layer with 256 input features and 10 output units (for CIFAR-10 classes). Softmax activation is implicitly applied by `nn.CrossEntropyLoss`.

Table 3.1: SimpleCNN Architecture Details

Layer	Input Shape	Output Shape	Kernel/Pool	Activation
Input	$3 \times 32 \times 32$	$3 \times 32 \times 32$	N/A	N/A
Conv1	$3 \times 32 \times 32$	$32 \times 32 \times 32$	3×3	SiLU
MaxPool1	$32 \times 32 \times 32$	$32 \times 16 \times 16$	2×2 (stride 2)	N/A
Conv2	$32 \times 16 \times 16$	$64 \times 16 \times 16$	3×3	SiLU
MaxPool2	$64 \times 16 \times 16$	$64 \times 8 \times 8$	2×2 (stride 2)	N/A
Conv3	$64 \times 8 \times 8$	$128 \times 8 \times 8$	3×3	SiLU
MaxPool3	$128 \times 8 \times 8$	$128 \times 4 \times 4$	2×2 (stride 2)	N/A
Flatten	$128 \times 4 \times 4$	2048	N/A	N/A
FC1	2048	256	N/A	SiLU
FC2	256	10	N/A	Softmax (implicit)

The SiLU (Swish) activation, $\text{SiLU}(x) = x \cdot \sigma(x)$, was consistently applied after each conv and the first FC layer.

3.3 Parameterized Weight Initialization

The study focused on adapting the widely used He initialization scheme [2] for networks using Swish/SiLU activations. He initialization, designed for ReLU-like activations, samples weights from a normal distribution $N(0, \sigma_{\text{He}}^2)$ where `fan_in` represents the number of input units to the layer and the standard deviation

$$\sigma_{\text{He}} = \sqrt{\frac{2.0}{\text{fan\_in}}}$$

A scaling factor s was introduced:

$$\sigma_{\text{scaled}} = s \cdot \sqrt{\frac{2.0}{\text{fan\_in}}} \quad \text{(Equation 3.1)} \tag{3.1}$$

The `init_weights` method applied this logic for both convolutional and linear layers:

- The value of `fan_in` is computed based on the layer type.
- The standard deviation is scaled by the factor s.
- Weights are initialized from a normal distribution, and biases are set to zero.

The RL agent explored values of s within a predefined range

3.4 Reinforcement Learning Environment (ScaleEnv)

A custom environment, `ScaleEnv`, compatible with the Gymnasium API [20], was developed to facilitate the interaction between the RL agent and the child network evaluation process.

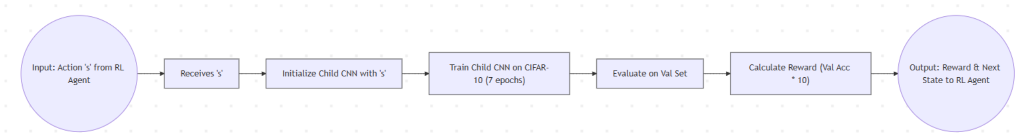

Figure 3.2: Workflow of the ScaleEnv environment, illustrating the interaction between the RL agent and the child network evaluation process.

3.4.1 State Space (`self.observation_space`)

The state provided to the RL agent at the beginning of each episode consisted of the previous scaling factor attempted (`self.current_scale`) and the reward obtained from that attempt (`self.current_reward`). The observation space was defined as a `spaces.Box` with:

- Low values: [`min_scale_explored`, -2.0] (where -2.0 is the NaN penalty).
- High values: [`max_scale_explored`, 10.0] (where 10.0 is the max possible reward if accuracy is 1.0 and scaled by 10).
- Shape: (2,) representing the two continuous values.
- Data type: `np.float32`.

Upon reset, `self.current_scale` was initialized by sampling uniformly from [`min_scale`, `max_scale`] and `self.current_reward` was set to 0.0.

3.4.2 Action Space (`self.action_space`)

The action space was continuous, representing the scaling factor s to be evaluated. It was defined as a `spaces.Box` with:

- Low value: `min_scale` (e.g., 0.1 or 0.3 as per `run_rl_training` parameters).
- High value: `max_scale` (e.g., 1.7 or 1.9 as per `run_rl_training` parameters).
- Shape: (1,).
- Data type: `np.float32`.

The raw action output by the agent's policy network was clipped to ensure it remained within [`min_scale`, `max_scale`] using `np.clip(action[0], self.min_scale, self.max_scale)` within the `step` method.

3.4.3 Evaluation Proxy Task (`train_and_evaluate` function)

Upon receiving an action (a proposed `scale_factor`), the `step` method of the environment instantiated the `SimpleCNN` with this `scale_factor`. This child network was then trained and evaluated using the `train_and_evaluate` function:

- Dataset: CIFAR-10.
- Training Epochs: A fixed number (`num_epochs`), typically 5 or 7 for the proxy task during RL agent training (parameter `num_epochs` in `ScaleEnv` and `run_rl_training`).
- Optimizer: Adam [21] with a learning rate specified (e.g., `lr=0.001`).
- Loss Function: `nn.CrossEntropyLoss`.
- Batch Size: 128.

The `train_and_evaluate` function returned the final validation accuracy on the CIFAR-10 test set and a boolean flag indicating if NaNs were encountered during training.

3.4.4 Reward Function

The reward R returned to the agent was based on the child network's 7-epoch (or proxy task epoch count) validation accuracy:

- $R = \text{validation\_accuracy} \times 10$ if no NaN was detected during training.
- $R = -2.0$ if a NaN was detected (e.g., `torch.isnan(loss)` or `torch.isinf(loss)`).

This corresponds to the logic in the `step` method of `ScaleEnv` and matches Equation 2 from the pilot study. The accuracy was scaled by 10 to provide a more substantial reward signal for the policy gradient updates. The NaN penalty strongly discouraged initializations leading to unstable training.

3.4.5 Episode Definition

Each episode consisted of a single step:

- The agent selected an action (`scale_factor`).
- The environment evaluated this `scale_factor` by training the child CNN for the proxy task duration.
- The environment returned the next state (updated `current_scale` and `current_reward`), the computed reward, a done flag (always `True` as episodes were single-step), and an info dictionary containing `scale_factor`, `accuracy`, and `nan_detected`.

3.5 Reinforcement Learning Agent (ReinforceAgent)

A REINFORCE policy gradient agent [22, 23] was implemented to learn the policy for selecting the scaling factor s.

3.5.1 Policy Network Architecture

The policy $\pi(a|s;\theta)$ was represented by a simple Multi-Layer Perceptron (MLP), defined within the `ReinforceAgent` class:

- Input Layer: `state_dim` units (2 units: previous scale, previous reward).
- Hidden Layer 1: 64 units, followed by ReLU activation.
- Hidden Layer 2: 64 units, followed by ReLU activation.
- Output Layer: `action_dim` $\times$ 2 units (i.e., 2 units for a 1D action space). These two units represented the mean (μ) and the logarithm of the standard deviation (`log_std`) of a Gaussian distribution from which the raw action was sampled.

3.5.2 Action Selection (`select_action` method)

- The current state was fed into the policy network to obtain μ and `log_std`.
- `log_std` was clamped (e.g., min=-20, max=2) to ensure numerical stability.
- The standard deviation `std` was computed as `log_std.exp()`.
- A raw action was sampled from the Normal distribution $N(\mu, \mathrm{std}^2)$.
- This raw action was then transformed to the desired range [`min_scale`, `max_scale`] using a sigmoid function to map it to [0, 1], followed by scaling and shifting: `scaled_action = torch.sigmoid(action) * self.action_range + self.min_scale`.
- The log probability of the sampled (pre-scaled) action, `dist.log_prob(action)`, was saved for the policy update.

3.5.3 Policy Update (`update` method)

The policy parameters θ were updated after each episode using the REINFORCE algorithm:

- **Calculate Returns:** Since episodes were single-step, the return G_t for the single step was simply the reward R obtained (γ was defined but effectively not used for a single-step return $R = r + \gamma \times 0$). The code in `update` calculates discounted returns $R = r + \gamma \times R_{\text{next}}$, which is standard REINFORCE. For single-step episodes, this simplifies to $G_0 = R_0$.
- **Normalize Returns (Optional but implemented):** If more than one reward was collected (though not in this single-step per episode setup for G_t, but over a batch of episodes if updates were batched), returns were normalized by subtracting the mean and dividing by the standard deviation (plus a small epsilon for stability). Your `update` method collects rewards from multiple episodes before an update if `agent.update()` is called less frequently than `agent.rewards.append()`. The pilot description implies update per episode. Clarify if update is after each single-step episode or batched.
- **Compute Policy Loss:** The loss was calculated as $L(\theta) = -\sum \log \pi_\theta(a_t|s_t) \times G_t$. In the code, this is `policy_loss.append(-log_prob * R)`.
- **Optimization:** The Adam optimizer was used to update the policy network parameters by performing gradient ascent on the expected return (or gradient descent on the negative loss).
- **Gradient Clipping:** Gradient norms were clipped (`torch.nn.utils.clip_grad_norm_`) to a maximum value (e.g., 1.0) to prevent exploding gradients during policy updates.

- Saved log probabilities and rewards were cleared after the update.

This detailed methodology provided a systematic way to explore the impact of the scaling factor s and allow the RL agent to learn an effective strategy for its selection.

Chapter 4

EXPERIMENTAL SETUP and DESIGN

This chapter outlines the experimental framework and procedures used to evaluate the proposed RL-based weight initialization strategy. It describes the datasets, training configurations for both the proxy and final validation models, RL agent setup, evaluation protocol, and computational tools and resources.

4.1 Dataset: CIFAR-10

The CIFAR-10 dataset [24] was used throughout this study.

- **Description:** 60,000 32×32 RGB images across 10 classes (50,000 training, 10,000 test).
- **Preprocessing (PyTorch proxy task):**
  - `ToTensor()`
  - `Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))` to scale pixel values to $[-1, 1]$.
- **Preprocessing (Keras final validation):**
  - Pixel values scaled to $[0, 1]$.
  - Labels one-hot encoded.

4.2 Child Network Training Configuration

4.2.1 RL Proxy Task (PyTorch SimpleCNN)

The RL loop trains a lightweight CNN called `SimpleCNN` using PyTorch. This model is used to evaluate the scaling factor s proposed by the RL agent.

- **Optimizer:** Adam [25], with learning rate 0.001 and default parameters.
- **Loss Function:** Cross-Entropy.
- **Batch Size:** 128.
- **Epochs:** Typically 5 or 7 (7 used in final proxy task).
- **Device:** Apple MacBook with M3 chip (MPS acceleration).
- **Initialization:** Weights initialized using a scaled He scheme:

$$\sigma_{\text{scaled}} = s \times \sqrt{\frac{2.0}{\text{fan\_in}}}$$

- **Reproducibility:** Random seeds fixed to 42; CUDA backends configured for deterministic behavior.

4.2.2 Final Validation Network (TensorFlow/Keras)

To validate the effectiveness of the scaling factor s, a deeper CNN was implemented in TensorFlow/Keras and trained for an extended duration.

Table 4.1: **Convolutional Neural Network Architecture Details**

Layer	**Filters**	**Input Shape**	**Output Shape**	**Pool Size**	**Activation**
Input	N/A	$3 \times 32 \times 32$	$3 \times 32 \times 32$	N/A	N/A
Conv Block 1					
Conv2D	32	$3 \times 32 \times 32$	$32 \times 32 \times 32$	3×3	Swish
BatchNorm	N/A	$32 \times 32 \times 32$	$32 \times 32 \times 32$	N/A	N/A
Conv2D	32	$32 \times 32 \times 32$	$32 \times 32 \times 32$	3×3	Swish
BatchNorm	N/A	$32 \times 32 \times 32$	$32 \times 32 \times 32$	N/A	N/A
MaxPooling2D	N/A	$32 \times 32 \times 32$	$32 \times 16 \times 16$	2×2 (stride 2)	N/A
Dropout	N/A	$32 \times 16 \times 16$	$32 \times 16 \times 16$	N/A	N/A
Conv Block 2					
Conv2D	64	$32 \times 16 \times 16$	$64 \times 16 \times 16$	3×3	Swish
BatchNorm	N/A	$64 \times 16 \times 16$	$64 \times 16 \times 16$	N/A	N/A
Conv2D	64	$64 \times 16 \times 16$	$64 \times 16 \times 16$	3×3	Swish
BatchNorm	N/A	$64 \times 16 \times 16$	$64 \times 16 \times 16$	N/A	N/A
MaxPooling2D	N/A	$64 \times 16 \times 16$	$64 \times 8 \times 8$	2×2 (stride 2)	N/A
Dropout	N/A	$64 \times 8 \times 8$	$64 \times 8 \times 8$	N/A	N/A
Conv Block 3					
Conv2D	128	$64 \times 8 \times 8$	$128 \times 8 \times 8$	3×3	Swish
BatchNorm	N/A	$128 \times 8 \times 8$	$128 \times 8 \times 8$	N/A	N/A
Conv2D	128	$128 \times 8 \times 8$	$128 \times 8 \times 8$	3×3	Swish
BatchNorm	N/A	$128 \times 8 \times 8$	$128 \times 8 \times 8$	N/A	N/A
MaxPooling2D	N/A	$128 \times 8 \times 8$	$128 \times 4 \times 4$	2×2 (stride 2)	N/A
Dropout	N/A	$128 \times 4 \times 4$	$128 \times 4 \times 4$	N/A	N/A
Dense Block					
Flatten	N/A	$128 \times 4 \times 4$	2048	N/A	N/A
Dense	512	2048	512	N/A	Swish
BatchNorm	N/A	512	512	N/A	N/A
Dropout	N/A	512	512	N/A	N/A
Output Layer	10	512	10	N/A	Softmax

Architecture:

Initialization: All Conv2D and Dense layers used `tf.keras.initializers.HeNormal()` with a custom scaling factor s (i.e., `ScaledHeNormal(scale=s)`).

Training Configuration:

- Optimizer: Adam (learning rate = 0.001)
- Loss: categorical_crossentropy
- Batch Size: 64
- Epochs: 100
- Device: NVIDIA T4 GPU (via cloud)

4.3 RL Agent and Environment Configuration

4.3.1 ReinforceAgent (Policy Gradient)

- **Policy Network:** As described in Section 3.5.1.
- **Optimizer:** Adam, learning rate 0.001.
- **Discount Factor:** $\gamma = 0.99$
- **Action Range:** $s \in [0.3, 1.7]$ for the final reported experiments.

4.3.2 Environment (ScaleEnv)

- **Observation Space:** Previous scale and reward.
- **Action Space:** Continuous scalar for s.
- **Reward:** $10 \times$ validation accuracy or -2.0 if NaN occurred.
- **Training per Episode:** 7 epochs on CIFAR-10.

4.4 Evaluation Protocol

4.4.1 Phase 1: RL Agent Training and Search

The RL agent interacted with `ScaleEnv` over 200 episodes. In each episode:

- A scaling factor s was proposed.
- A child CNN was initialized with s and trained for 7 epochs.
- Validation accuracy was returned as reward.
- The best non-NaN s (denoted s_{proxy}) was selected based on maximum reward.

Metrics including rewards, accuracies, and NaN occurrences were logged via Weights & Biases [26].

4.4.2 Phase 2: Final Validation with Keras CNN

To evaluate the generalizability of s_{proxy}, the deeper TensorFlow CNN (Section 4.2.2) was trained twice for 100 epochs:

- Once using $s = 1.0$ (standard He initialization)
- Once using $s = s_{\text{proxy}}$ (e.g., 1.25)

Final test accuracies were compared to assess performance gain from the RL-optimized initialization scale.

4.5 Tools and Libraries

- **Language:** Python 3.
- **Libraries:**
  - PyTorch [27]
  - TensorFlow/Keras
  - NumPy [28]
  - Gymnasium [29]
  - Matplotlib [30]
  - tqdm
- **Experiment Tracking:** Weights & Biases [26]

4.6 Computational Resources

- **RL Training (Proxy):** Apple MacBook with M3 chip using MPS backend.
- **Final Validation:** NVIDIA T4 GPU via cloud environment.
- **Device Detection:** The code auto-selects between MPS, CUDA, and CPU depending on hardware availability.

Chapter 5

RESULT and ANALYSIS

5.1 Introduction to Experimental Results

This chapter presents the empirical results obtained from the Reinforcement Learning (RL) framework designed to tune the initialization scaling factor s for He initialization when using Swish activation functions. As detailed in Chapter 3 (Methodology), an RL agent was tasked with exploring a continuous range of s values (specifically $[0.3, 1.7]$ in the final documented experiment), receiving rewards based on the 7-epoch validation accuracy of a small Convolutional Neural Network (SimpleCNN in PyTorch, hereafter referred to as the "proxy network") trained on the CIFAR-10 dataset. The primary objectives were to assess the RL agent's learning capability and to identify a potentially optimal s for this 7-epoch proxy task using the proxy network. Subsequently, the characteristics of the discovered scaling factor region were validated through more extensive training (100 epochs) using a different, more standard, and robust CNN architecture implemented in TensorFlow/Keras (hereafter referred to as the "validation network") to evaluate the generalizability of the RL-identified scaling factor to a distinct and more complex model.

5.2 Reinforcement Learning Agent Training Dynamics with the Proxy Network

The REINFORCE agent was trained for a total of 200 episodes. Each episode involved the agent selecting a scaling factor s, initializing the proxy network (SimpleCNN), training it for 7 epochs, and receiving a reward based on the resulting validation accuracy. The evolution of key metrics during this training process was logged using Weights & Biases [**?**] and is presented in Figure 5.1.

5.2.1 Proxy Network Performance During RL Search

The top row of Figure 5.1 illustrates the performance of the proxy network (SimpleCNN) during each of the 200 RL episodes. The `train_loss` and `train_acc` plots show the characteristic learning curves for a 7-epoch training run on this proxy network, repeated for each new scaling factor proposed by the agent. The considerable variance in these plots is expected, as each of the 200 "steps" on the x-axis represents the aggregated metrics from a distinct 7-epoch training session of the proxy network with potentially different initialization scales. These plots confirm that the proxy network was generally trainable across the range of scaling factors explored. The `test_accuracy` plot (top-right), which directly informs the agent's reward (scaled by 10), shows values fluctuating primarily between approximately 0.70 and 0.75 for the proxy network. There appears to be a slight upward drift in the average and peak test accuracy achieved as the RL training progresses, particularly in the latter half of the episodes.

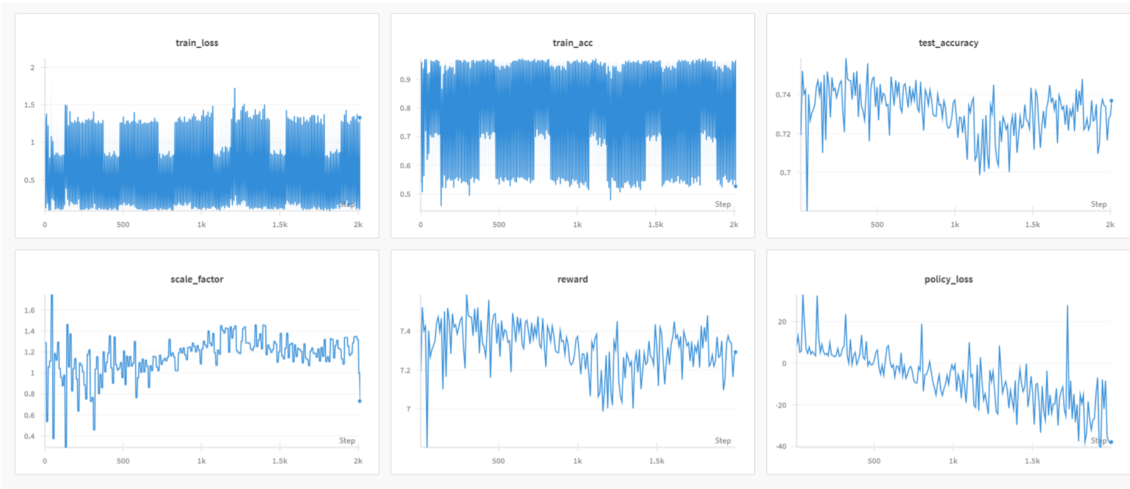

Figure 5.1: Reinforcement Learning search dynamics over 200 episodes using the SimpleCNN proxy network. (Top Row, L-R) Proxy network training loss per RL episode, proxy network training accuracy per RL episode, proxy network test accuracy (reward/10) per RL episode. (Bottom Row, L-R) Sampled scale_factor by the RL agent, received reward by the RL agent, and RL agent's policy_loss.

5.2.2 RL Agent's Action Selection (Scaling Factor s)

The `scale_factor` plot (Figure 5.1, bottom-left) depicts the evolution of the s values sampled by the RL agent for the proxy network. Initially, during the first ∼250–300 episodes, the agent exhibited broad exploration, sampling s values across a wide segment of its allowed range (approximately 0.3 to 1.7). Following this initial exploratory phase, the agent's sampling behavior began to converge. From approximately episode 500 onwards, the agent predominantly focused its exploration on s values within a narrower band, roughly between 1.1 and 1.4. This shift from broad exploration to more focused exploitation of a specific region indicates that the agent identified this range as generally yielding higher rewards for the 7-epoch proxy task when using the SimpleCNN. While some exploration outside this band persisted, the mean of the sampled s values clearly stabilized in this higher range compared to the initial phase.

5.2.3 RL Agent's Reward and Policy Loss

The reward plot (Figure 5.1, bottom-middle) mirrors the test accuracy of the proxy network. Consistent with the test accuracy observations, the reward signal exhibits significant variance but suggests a marginal increase in the average reward received by the agent as it learned to favor the identified range of scaling factors for the proxy network. The policy loss for the REINFORCE agent (Figure 5.1, bottom-right) shows a distinct, albeit noisy, downward trend over the 200 episodes. This reduction in policy loss is a strong indicator of policy improvement and stabilization, suggesting that the agent became more confident in its action selections as training progressed.

5.3 Optimal Scaling Factor for the 7-Epoch Proxy Task using SimpleCNN

Based on the 200-episode RL search conducted with the SimpleCNN proxy network, the agent effectively learned to explore the scaling factor space for the 7-epoch validation accuracy. The region of scaling factors $s \in [1.1, 1.4]$ was identified as consistently yielding higher rewards when applied to the SimpleCNN.

The single best performing scaling factor encountered during the entire 200-episode search for the SimpleCNN, denoted as s_{proxy} (or $s_{\text{proxy\_best\_7\_epoch\_run}}$), was 1.25. With this scaling factor, the SimpleCNN proxy network achieved a 7-epoch validation accuracy of $\text{Accuracy}(s_{\text{proxy}} = 1.25, \text{SimpleCNN}, 7 \text{ epochs}) = 0.7367$.

For comparison, when the baseline He initialization ($s = 1.0$) was applied to the SimpleCNN proxy network, it achieved a 7-epoch validation accuracy of $\text{Accuracy}(s = 1.0, \text{SimpleCNN}, 7 \text{ epochs}) = 0.7369$.

In this specific 200-episode search with the SimpleCNN, the absolute best 7-epoch performance was achieved by the baseline $s = 1.0$, albeit by a very small margin (0.0002). The RL agent converged to a region (1.1–1.4) near this, with its best single discovery ($s_{\text{proxy}} = 1.25$) performing almost identically on the SimpleCNN. No instances of numerical instability (NaNs) were frequently reported in the preferred range of s, indicating that the explored range and the proxy network/training setup were generally stable for these short 7-epoch runs.

5.4 Validation of Discovered Scaling Factor on Extended Training with a Standard TensorFlow/Keras CNN

While the RL agent identified $s_{\text{proxy}} = 1.25$ as a promising scaling factor based on its performance with the simpler SimpleCNN proxy network in the 7-epoch task (Section 5.3), a more rigorous and critical test is its generalizability to longer training durations and, importantly, to a different, more standard, and robust model architecture. Therefore, to assess this transferability, the efficacy of applying $s_{\text{proxy}} = 1.25$ was evaluated by training a distinct Convolutional Neural Network, implemented in TensorFlow/Keras (as detailed in Section 4.5 and referred to as the "validation network"), for an extended period of 100 epochs on the CIFAR-10 dataset. This validation network is more complex than the SimpleCNN used in the RL loop, featuring multiple VGG-style convolutional blocks incorporating Swish activation, Batch Normalization, MaxPooling, and Dropout layers, followed by dense layers. The performance of this validation network initialized with `initializer_scale` $= s_{\text{proxy}} = 1.25$ was compared against the same validation network initialized with the baseline He scaling factor (`initializer_scale` $= 1.0$). All other training parameters for this extended validation, such as optimizer (Adam, `learning_rate`=0.001), loss function (`categorical_crossentropy`), and batch size (64), were kept consistent for both $s = 1.0$ and $s = 1.25$ runs on the validation network.

The final validation accuracies on the CIFAR-10 test set after 100 epochs of training the TensorFlow/Keras validation network are summarized in Table 5.1.

Table 5.1: Validation Accuracy After 100 Epochs

Scaling Factor (s)	**Test Accuracy**	**Test Accuracy Run 2**
$s = 1.0$ (Baseline He)	0.8606	0.8578
$s = s_{\text{proxy}}$ (1.25)	0.8642	0.8616

The results in Table 5.1 indicate that the scaling factor $s_{\text{proxy}} = 1.25$ maintained, and indeed slightly extended, its advantage over the baseline He initialization ($s = 1.0$) when training was carried out for 100 epochs. On average, $s_{\text{proxy}} = 1.25$ achieved a validation accuracy of 0.8629, compared to 0.8592 for $s = 1.0$. This represents an average improvement of approximately 0.0037, or 0.37 percentage points. While modest, this improvement suggests that the signal captured by the RL agent from the 7-epoch proxy task was directionally correct and translated to tangible benefits in longer training. The consistency across two runs (Run 1: +0.0036, Run

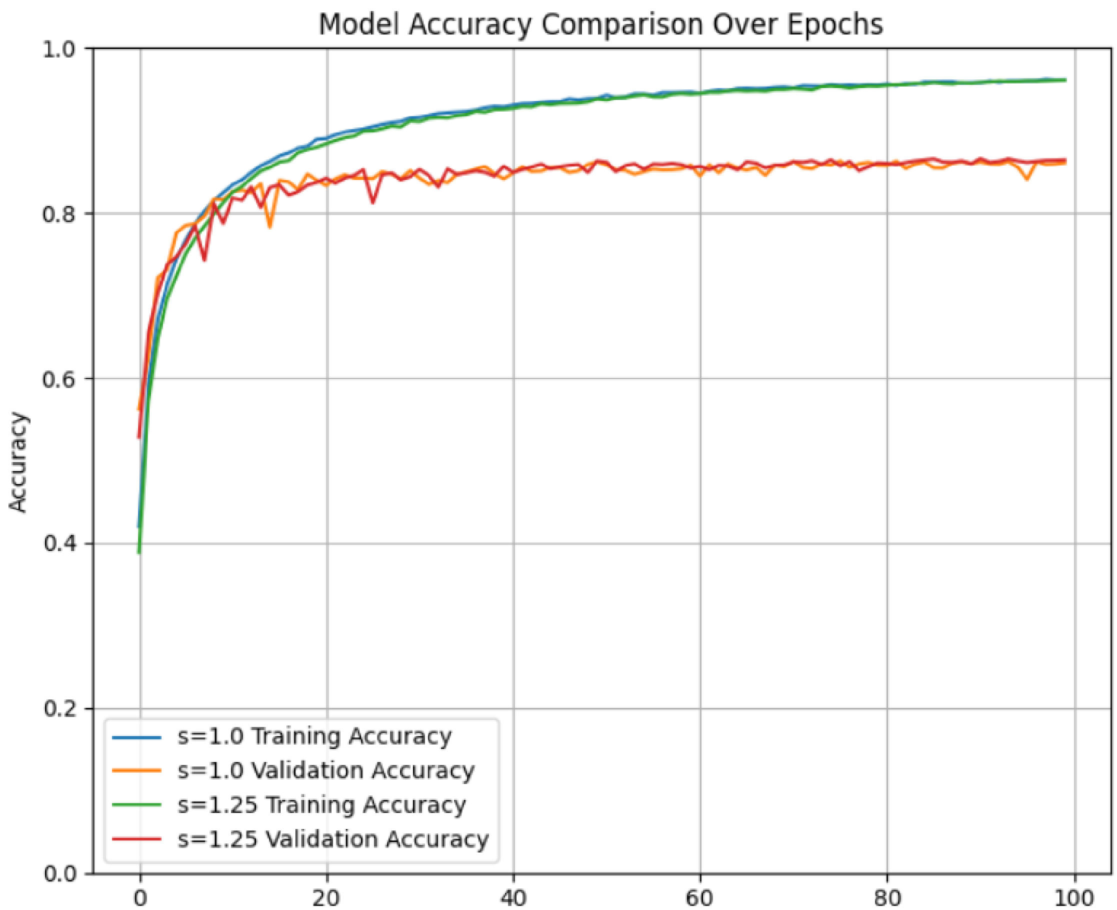

Figure 5.2: Training/validation accuracy and loss curves for the TensorFlow/Keras validation CNN over 100 epochs, comparing baseline He initialization ($s = 1.0$) and the RL-identified scaling factor ($s_{\text{proxy}} = 1.25$).

2: +0.0038) lends further support to this observation, although more runs would be beneficial for stronger statistical claims.

5.4.1 Analysis of Learning Curves

Analysis of the learning curves shown in Figure 5.2 reveals several key insights:

- **Initial Learning Phase (Epochs 1–20):** Observing the initial phase of training, both $s = 1.0$ and $s = 1.25$ exhibit rapid increases in training and validation accuracy.

- **Mid-to-Late Training Phase (Epochs 20–100):** As training progresses, both configurations appear to converge. The validation accuracy for $s = 1.25$ consistently tracks slightly above that of $s = 1.0$ throughout a significant portion of this phase, culminating in the higher final validation accuracy reported in Table 5.1.

- **Training Accuracy vs. Validation Accuracy:** Both configurations show training accuracy surpassing validation accuracy, indicative of some level of overfitting, which is common. A key point of comparison would be whether the gap between training and validation accuracy (generalization gap) differs significantly between the two initialization strategies. The current plot suggests comparable generalization gaps, but a plot of the loss curves (both training and validation) might provide additional insights into overfitting and optimization stability.

- **Stability:** The curves for $s = 1.25$ appear to be as smooth as, if not slightly smoother than, those for $s = 1.0$, suggesting that the modified initialization did not introduce instability into the training process.

The training accuracy for both configurations on the TensorFlow/Keras validation network surpassed their respective validation accuracies, indicating some degree of overfitting, which is typical for such models on CIFAR-10 even with regularization like Dropout and Batch Normalization. The generalization gap (difference between training and validation accuracy) appeared comparable for both scaling factors. The smoothness of the learning curves for $s_{\text{proxy}} = 1.25$ relative to $s = 1.0$ suggests that the chosen scaling factor did not introduce additional instability into the training process of the validation network.

This rigorous validation on a distinct and more complex architecture provides crucial context for the RL agent's findings. It underscores that while an RL agent can effectively optimize for a given proxy task and network, the direct transferability of the exact discovered parameters to different architectural settings requires careful empirical verification.

5.5 Summary of Experimental Findings

The Reinforcement Learning search mechanism, operating on a SimpleCNN proxy network for 7-epoch evaluations, successfully demonstrated learning. It converged its exploration of the initialization scaling factor s towards a region (approximately $s \in [1.1, 1.4]$) that generally yielded high 7-epoch validation accuracy for this proxy network. The agent's policy loss also showed a consistent decrease, indicating policy improvement. The best single scaling factor encountered by the agent during this proxy search was $s_{\text{proxy}} = 1.25$, achieving an accuracy of 0.7367 on the SimpleCNN, which was very close to the baseline $s = 1.0$ (0.7369) on the same proxy network.

A more critical test involved applying this RL-identified $s_{\text{proxy}} = 1.25$ to a different, more complex TensorFlow/Keras validation network for an extended 100-epoch training period. On this validation network, $s_{\text{proxy}} = 1.25$ achieved a test accuracy of 0.8629, while the baseline $s = 1.0$ achieved average test accuracy of 0.8592. This outcome provides important context for interpreting the results of automated search methods that rely on proxy tasks and networks, highlighting the potential for generalizable discovery.

Chapter 6

DISCUSSION

6.1 Interpretation of RL Agent Behavior and Discovered Parameters

The 200-episode Reinforcement Learning experiment, conducted with the `SimpleCNN` proxy network, provided valuable insights into the agent's learning process and the nature of the optimization landscape for the initialization scaling factor s within that specific context. The agent's observable shift from broad exploration of s values (spanning approximately 0.3 to 1.7) to a more focused sampling regimen concentrated around values between 1.1 and 1.4 (as depicted in Figure 5.1) unequivocally indicates that it successfully identified this particular region as yielding higher rewards within the 7-epoch evaluation framework of the proxy network. This convergence, when considered alongside the consistent decrease in the agent's policy loss, serves as strong confirmation of the REINFORCE algorithm's efficacy in navigating and optimizing within this specific, albeit simplified, parameter tuning context for the `SimpleCNN`.

A particularly intriguing observation is the fact that the agent favored $s > 1.0$ (e.g., the 1.1–1.4 region) for the 7-epoch proxy task with the `SimpleCNN`, even though the absolute best single 7-epoch run within the 200 episodes was achieved by $s = 1.0$ (albeit by a very marginal difference of 0.0002). This preference suggests that, for the `SimpleCNN` architecture employing Swish activations, scaling factors slightly larger than standard He initialization might confer an advantage in the very early stages of training. This could manifest as an initial acceleration in learning or a more effective means of overcoming initial learning plateaus over a short training horizon like 7 epochs. The Swish activation function, with its non-monotonic nature and regions where its derivative can be relatively small, might benefit from slightly larger initial weight magnitudes to ensure sufficient signal strength and robust gradient propagation during these crucial initial iterations. The RL agent, through a process of purely empirical trial and error on the `SimpleCNN`, effectively "discovered" this locally beneficial region for rapid, short-term performance.

6.2 Fidelity of the 7-Epoch Proxy Task and Transferability of Findings

A central theme emerging from this study is the fidelity of the short-term proxy task (7-epoch training of the `SimpleCNN`) in predicting longer-term performance, especially when the insights are transferred to a different, more complex validation network. The results from the extended 100-epoch validation using the TensorFlow/Keras validation network (detailed in Section 5.4) are critical in this assessment.

The observation that $s_{\text{proxy}} = 1.25$ (a representative value from the RL-favored region identified using the `SimpleCNN` proxy) maintained and indeed slightly improved its performance advantage over the baseline $s = 1.0$ when both were applied to the distinct TensorFlow/Keras validation network and trained for 100 epochs is a highly encouraging and significant result.

Specifically, the TensorFlow/Keras validation network achieved an average test accuracy of 0.8629 with $s_{\text{proxy}} = 1.25$, compared to 0.8592 with $s = 1.0$. This outcome suggests that, for this specific problem of tuning a single scaling factor for He initialization with Swish, the 7-epoch proxy task conducted on the simpler `SimpleCNN` provided a remarkably reliable signal that generalized positively not only to longer training durations but also across a notable architectural shift to the more complex TensorFlow/Keras validation network.

While it's true that $s = 1.0$ yielded a marginally higher single-best accuracy on the 7-epoch `SimpleCNN` proxy task itself, the RL agent's convergence to the 1.1–1.4 region (which included $s_{\text{proxy}} = 1.25$) proved to be a strategically sound discovery for long-term performance on the more robust validation architecture. This could be attributed to the fundamental role of the initialization scaling factor in influencing the very initial dynamics of training. If an s value establishes a favorable learning trajectory early on—one that promotes stable and efficient gradient flow without leading to immediate instability—that benefit can persist and compound over more extended training, even in a different network that shares the same core activation function (Swish) and initialization strategy (scaled He). The fact that this positive correlation held despite the architectural differences (including the presence of Batch Normalization and Dropout in the TensorFlow/Keras validation network) strengthens the finding. However, it remains crucial to acknowledge that such successful transfer might not universally hold for more disparate architectural changes, vastly different tasks, or when searching for more complex, multi-parameter initialization formulas.

6.3 Comparison to Standard Initialization and Implications for Swish Activation

The consistent exploration by the RL agent of s values often greater than 1.0 during the 7-epoch proxy task with the `SimpleCNN` is an intriguing finding, especially when considering the origins of standard He initialization. He initialization was primarily derived under the assumption of ReLU-like activation functions. The Swish activation function, however, possesses distinct mathematical properties, including its non-monotonicity and smooth, self-gated behavior. It is therefore plausible that the "ideal" initial weight variance for networks employing Swish might indeed differ, even if subtly, from that prescribed for ReLU. This pilot study, by empowering the RL agent to search for an optimal scaling factor for He initialization when used with Swish, provides a data-driven, albeit indirect, exploration of this hypothesis.

The subsequent superior performance of $s_{\text{proxy}} = 1.25$ in the extended 100-epoch training of the more complex TensorFlow/Keras validation network provides compelling empirical evidence. It suggests that a simple, modest scaling of the standard He initialization (specifically, increasing the variance slightly) can be advantageous when Swish activations are used, at least within the context of the CNN architectures and the CIFAR-10 dataset investigated here. This implies that for Swish, a slightly larger initial variance than that dictated by the standard He formula ($s = 1.0$) might be beneficial for achieving better final model performance, possibly by ensuring a more robust initial signal propagation or by helping the optimization navigate the early loss landscape more effectively.

6.4 Limitations of the Study

This preliminary investigation, while yielding insightful results, carries several limitations that must be acknowledged when interpreting its findings and considering their broader implications:

- **Architectural Discrepancy in Proxy Task Fidelity:** While the 7-epoch proxy task using the `SimpleCNN` successfully guided the RL agent to an s_{proxy} value that generalized well to the more complex TensorFlow/Keras validation network, this positive outcome might not be universally guaranteed. The degree of architectural mismatch between a proxy network and a target validation network can significantly impact the transferability of discovered hyperparameters. A more direct, albeit computationally expensive, approach would involve using a scaled-down version of the target architecture within the RL loop.

- **Simplified Search Space:** The exploration was deliberately constrained to a single scalar parameter s modifying a fixed He initialization structure. This represents a highly simplified search space compared to the ultimate research goal of discovering novel, potentially complex, symbolic initialization formulas. The optimal s found is conditional upon the underlying He structure and might change if the base formula itself were different.

- **Specificity of Network Architectures and Dataset:** The results obtained are specific to the `SimpleCNN` (PyTorch) used for the RL search, the more complex CNN (TensorFlow/Keras) used for validation, and the CIFAR-10 dataset. The optimal scaling factor s, and indeed the fidelity of any given proxy task, could vary significantly for different network architectures (e.g., much deeper ResNets, Transformers), other datasets (e.g., ImageNet), or if the Swish hyperparameter β were different or a learnable parameter.

- **RL Algorithm Choice and Hyperparameter Tuning:** The REINFORCE algorithm, while foundational for policy gradients, is known for its high sample variance and can be less sample-efficient than more advanced alternatives. Algorithms such as PPO (Proximal Policy Optimization) or SAC (Soft Actor-Critic) might offer improved learning stability, faster convergence, or the ability to find even better solutions. Furthermore, the hyperparameters for the REINFORCE agent itself (e.g., learning rate, policy network architecture) were chosen based on common practices and were not exhaustively tuned for this specific initialization problem.

- **Reward Signal Simplicity:** The reward signal for the RL agent was predominantly based on the 7-epoch validation accuracy achieved by the proxy network, with a strong penalty for NaN occurrences. This reward function did not explicitly incorporate more nuanced metrics of training stability (such as the variance of gradient norms, statistics of activation distributions across layers) or the speed of convergence in the early epochs, which could potentially guide the agent towards solutions with even better long-term properties or improved robustness.

- **Single RL Search Run:** The 200-episode RL search, while extensive for a pilot, represents a single training run of the agent. To rigorously assess the variance and consistency of the discovered optimal s value and the characteristics of the converged region, multiple independent RL search runs would ideally be conducted. (The presence of two runs for the 100-epoch validation of the *final* s values is a good practice for assessing the stability of the final model training, but does not address the RL search variance itself).

6.5 Implications for Broader Research on Automated Initialization Discovery

Despite its acknowledged limitations, this pilot study offers valuable takeaways that directly inform the main dissertation work and contribute to the broader field of automated discovery for neural network initialization:

1. **Demonstrated Feasibility of RL for Initialization Tuning:** The study confirms that Reinforcement Learning agents can indeed be effectively trained to optimize continuous parameters related to neural network initialization schemes based on empirical performance feedback from training proxy networks.

2. **Criticality of Evaluation Strategy and Proxy Fidelity:** The nuanced relationship observed between the short-term proxy performance (7-epochs on `SimpleCNN`) and the longer-term validation on a different, more complex network (100-epochs on TensorFlow/Keras CNN) starkly illustrates the paramount importance of careful design and understanding of evaluation protocols within any automated search paradigm. When short-duration or simplified proxy evaluations are employed, their correlation with the true, ultimate objective must be rigorously assessed, or strategies to mitigate the potential "proxy gap" must be actively considered. The finding that the RL-favored region for the proxy led to a beneficial s_{proxy} for the validation network, even though the proxy optimum itself was subtly different, is an important subtlety highlighting that proxies can guide towards generally good regions.

3. **Potential for More Sophisticated Reward Signals:** The reliance solely on short-term accuracy, while functional, might be insufficient for discovering initializations that are optimal across a broader range of desirable characteristics (e.g., stability, faster convergence to good solutions, better generalization). Future research, including the planned symbolic search in this dissertation, should give serious consideration to incorporating more direct measures of training stability and learning dynamics into the reward function provided to the RL agent.

4. **Highlighting Computational Considerations:** The process of tuning even a single continuous parameter (s) required a substantial number of child network trainings (200 episodes $\times$ 7 epochs per episode). The subsequent, more ambitious goal of searching for complex symbolic initialization formulas will undoubtedly be orders of magnitude more computationally demanding. This underscores the critical need for highly efficient search strategies, robust and informative (yet inexpensive) evaluation proxies, and potentially more sample-efficient RL algorithms.

This pilot study has thus effectively served its intended purpose. It has provided a functional baseline RL framework for initialization tuning, brought key challenges and considerations—particularly regarding proxy task design and evaluation fidelity across different architectures—into sharp focus, and has been instrumental in refining the research questions and methodological approaches for the subsequent, more comprehensive exploration of discovering novel symbolic initialization formulas.

Chapter 7

CONCLUSION AND FUTURE SCOPE

7.1 Summary of Findings from the Study

This dissertation included a preliminary investigation into the use of Reinforcement Learning (RL) for tuning a scaling factor s for He initialization, specifically when employing Swish activation functions. The key findings from this pilot study, which involved an RL search using a `SimpleCNN` (PyTorch) proxy network and subsequent validation on a distinct, more complex TensorFlow/Keras validation network, are:

1. **RL Agent Learning on Proxy Network:** An RL agent, trained for 200 episodes using the `SimpleCNN` proxy network, successfully learned to optimize the scaling factor s based on a reward signal derived from 7-epoch validation accuracy. The agent demonstrated convergence by focusing its exploration on a specific range of s values (approximately $s \in [1.1, 1.4]$) for this proxy network and exhibited a decreasing policy loss, indicating policy improvement.

2. **Optimal Parameter for Proxy Task with `SimpleCNN`:** The agent identified $s_{\text{proxy}} = 1.25$ as the best single scaling factor for maximizing 7-epoch validation accuracy from its search on the `SimpleCNN` proxy network, achieving an accuracy of 0.7367. This was marginally lower than the baseline He initialization ($s = 1.0$) which achieved 0.7369 under the same 7-epoch conditions with the `SimpleCNN`.

3. **Generalizability to Extended Training on a Different, More Complex Validation Network:** The critical validation of $s_{\text{proxy}} = 1.25$ was performed by applying it to a distinct, more complex TensorFlow/Keras validation network for an extended training duration of 100 epochs. On this validation network, the RL-favored scaling factor ($s_{\text{proxy}} = 1.25$) resulted in an average test accuracy of 0.8629, outperforming the baseline $s = 1.0$ which achieved an average test accuracy of 0.8592 on the same TensorFlow/Keras validation network.

4. **Nuances of Proxy Task Fidelity and Transferability:** The study highlighted the critical dependence of the discovered solution's utility on how well insights from a short-term evaluation proxy (7-epoch training on `SimpleCNN`) align with the desired outcome on a different target system (100-epoch training on the TensorFlow/Keras validation network). While the 7-epoch proxy with `SimpleCNN` did not perfectly rank $s_{\text{proxy}} = 1.25$ over $s = 1.0$ for its own short-term performance, it successfully guided the RL agent to a region that demonstrated superior long-term performance when transferred to the more complex validation network.

7.2 Key Contributions of the Study

Within the context of the broader dissertation, this pilot study made the following contributions:

- **Methodological Feasibility of RL for Initialization Tuning:** It established a functional RL framework capable of exploring and optimizing parameters related to neural network initialization, serving as a foundational component and proof-of-concept for more complex, symbolic searches.
- **Empirical Insights into Proxy Task Design and Transferability:** It provided concrete empirical evidence regarding the challenges and potential successes of relying on simplified proxy networks and short-duration tasks for guiding automated search. The successful transfer of a beneficial scaling factor from the `SimpleCNN` proxy to the more complex TensorFlow/Keras validation network, despite the architectural differences, is a valuable finding. The nuance that a generally good *region* can be found, even if the proxy optimum itself isn't perfectly aligned with the long-term target optimum, is particularly insightful.
- **Informed Design for Subsequent Research:** The lessons learned, particularly concerning evaluation fidelity across different architectures, the nature of the reward signal, and computational trade-offs, directly inform and refine the design choices for the main research endeavor focused on discovering novel symbolic initialization formulas.

7.3 Limitations (Recap from Discussion)

It is important to reiterate the key limitations of this specific pilot study, including the simplified search space (a single scaling factor), the specificity of the `SimpleCNN` proxy network and the TensorFlow/Keras validation network to the CIFAR-10 dataset, the inherent characteristics of the REINFORCE algorithm, the primary reliance on short-term accuracy from the proxy network for the reward signal, and the architectural mismatch between the proxy search network and the final validation network. These limitations frame the study as preliminary and exploratory, setting the stage for more comprehensive investigations.

7.4 Future Work and Main Dissertation Directions

This pilot study serves as a crucial stepping stone towards the primary research goals of this dissertation. The insights gained pave the way for several avenues of future work, which form the core of the subsequent chapters:

1. **Symbolic Initialization Formula Search:** The immediate next step is to expand the search space from a single continuous parameter to the discovery of complex, symbolic formulas for the initialization variance σ^2 (or standard deviation σ), as originally proposed. This will involve developing or adapting a more sophisticated RL controller or search mechanism capable of generating and evaluating structured sequences of mathematical primitives and input features (e.g., `fan_in`, `fan_out`).
2. **Enhanced Reward Engineering:** Based on the findings, future iterations of the RL search (particularly for symbolic formulas) will explore more nuanced and potentially multi-objective reward functions. This may include:
    - Incorporating direct metrics of training stability (e.g., gradient norm variance, smoothness of loss, statistics of activation distributions) from early epochs of the proxy evaluation as components of the reward or as explicit penalties.
    - Investigating metrics related to the shape, rate of decrease, or convergence properties of the training loss curve as potentially better predictors of final performance or generalization.

- Exploring methods to balance short-term performance on a proxy task with indicators of longer-term stability and robustness.

3. **Adaptive Evaluation Budgets and Proxy Architectures:** Investigate strategies for dynamically allocating computational budget during the search, perhaps training more promising candidates for longer or using techniques like learning curve extrapolation. Crucially, explore the use of proxy networks that, while still computationally cheaper, more closely mirror the key architectural characteristics of the target validation networks to improve the direct transferability of discovered initializers.

4. **Broader Architectural and Dataset Validation:** Any promising symbolic initializers discovered will require rigorous and extensive validation across a wider range of modern neural network architectures (including deeper models like various ResNet configurations, Vision Transformers) and more challenging, diverse datasets (e.g., ImageNet or its subsets, other computer vision tasks, or even tasks in different domains like NLP).

5. **Exploration of Different RL Algorithms or Search Methods:** For the computationally demanding symbolic search, consider employing more sample-efficient and stable RL algorithms (e.g., PPO, SAC) or exploring alternative black-box optimization and automated machine learning techniques (e.g., evolutionary algorithms, Bayesian optimization adapted for symbolic spaces).

6. **Theoretical Analysis of Discovered Formulas:** For any truly novel and demonstrably effective symbolic initialization formulas discovered, an attempt should be made at a theoretical analysis to understand the underlying principles of why they might be beneficial for specific activation functions, network structures, or data characteristics.

7.5 Concluding Remarks on the Study

In conclusion, the pilot study on tuning an initialization scaling factor via Reinforcement Learning successfully demonstrated the potential of such an automated approach and, critically, highlighted key challenges and considerations for future work. The direct outcome regarding an improved scaling factor for Swish was positive: the RL-favored $s_{\text{proxy}} = 1.25$ (found via a 7-epoch proxy task on a `SimpleCNN`) did indeed generalize to improved 100-epoch performance on a different, more complex TensorFlow/Keras validation network when compared to the standard $s = 1.0$ baseline. This successful transfer, despite architectural differences, is an encouraging result.

However, the methodological insights gained regarding proxy task design, reward engineering, and the nuances of evaluating transferability are arguably even more invaluable. This work affirms that while RL can be a powerful tool for navigating complex design spaces in deep learning, the careful construction of the evaluation environment (including the choice of proxy network and task duration) and the reward signal is paramount. These elements must be designed to ensure that the solutions discovered are not merely optimal for a contrived proxy but are genuinely beneficial for the ultimate goal of training effective, robust, and generalizable neural networks. The path is now clearer, and the groundwork more solidly laid, for embarking on the more ambitious search for novel, symbolic initialization paradigms.

Appendix A

CODE

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import numpy as np
import random
import matplotlib.pyplot as plt
import time
from tqdm import tqdm  # Changed from tqdm.notebook to regular tqdm
import os
import wandb
import gymnasium as gym
from gymnasium import spaces
from collections import deque

# Create data directory if it doesn't exist
os.makedirs('./data', exist_ok=True)

# Set seeds for reproducibility
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False

set_seed()

# Check if MPS (Apple Silicon GPU) is available
if torch.backends.mps.is_available():
    device = torch.device("mps")
    print(f"Using MPS (Apple Silicon GPU)")
elif torch.cuda.is_available():
    device = torch.device("cuda")
    print(f"Using CUDA")
else:
    device = torch.device("cpu")
    print(f"Using CPU")

# PHASE 1: CORE SETUP
```

```
# -------------------

# Child Network: Simple CNN with Swish activation
class SimpleCNN(nn.Module):
    def __init__(self, scale_factor=1.0):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 4 * 4, 256)
        self.fc2 = nn.Linear(256, 10)

        # Initialize with modified He initialization using scale_factor
        self.init_weights(scale_factor)

    def init_weights(self, scale_factor):
        # Apply scaled He initialization to all convolutional and linear layers
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                fan_in = m.weight.data.size()[1]  # fan_in
                if isinstance(m, nn.Conv2d):
                    fan_in *= m.kernel_size[0] * m.kernel_size[1]
                std = scale_factor * np.sqrt(2.0 / fan_in)
                nn.init.normal_(m.weight.data, mean=0.0, std=std)
                if m.bias is not None:
                    nn.init.constant_(m.bias.data, 0.0)

    def forward(self, x):
        x = F.silu(self.conv1(x))  # Using SiLU (Swish) activation
        x = self.pool(x)
        x = F.silu(self.conv2(x))
        x = self.pool(x)
        x = F.silu(self.conv3(x))
        x = self.pool(x)
        x = x.view(-1, 128 * 4 * 4)
        x = F.silu(self.fc1(x))
        x = self.fc2(x)
        return x

# Basic Training Function
def train_and_evaluate(scale_factor, num_epochs=10, batch_size=128, lr=0.001,
↪ log_wandb=False):
    """Train the CNN with given scale factor and return validation accuracy."""

    # Data Loading and Preprocessing
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    # Load CIFAR-10 dataset
    train_dataset = datasets.CIFAR10(root='./data', train=True, download=True,
    ↪ transform=transform)
    test_dataset = datasets.CIFAR10(root='./data', train=False, download=True,
    ↪ transform=transform)

```

```
    # Using num_workers=0 for better compatibility on macOS
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
    num_workers=0)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,
    num_workers=0)

    # Create model with the given scale factor
    model = SimpleCNN(scale_factor=scale_factor).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    nan_detected = False

    # Training loop
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        # Use tqdm for progress bar
        for inputs, labels in tqdm(train_loader, desc=f"Epoch
        {epoch+1}/{num_epochs}", leave=False):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Check for NaN
            if torch.isnan(loss) or torch.isinf(loss):
                nan_detected = True
                break

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        if nan_detected:
            break

        # Epoch statistics
        epoch_loss = running_loss / len(train_loader)
        epoch_acc = correct / total
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Train Acc:
        {epoch_acc:.4f}')

        # Log to wandb if requested
        if log_wandb:
            wandb.log({
                'epoch': epoch,
```

```python
                'train_loss': epoch_loss,
                'train_acc': epoch_acc,
                'scale_factor': scale_factor
            })

    # If NaN detected, return early with penalty
    if nan_detected:
        if log_wandb:
            wandb.log({'nan_detected': True, 'scale_factor': scale_factor})
        return -1.0, True

    # Evaluation
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in tqdm(test_loader, desc="Evaluating", leave=False):
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = correct / total
    print(f"Test Accuracy: {accuracy:.4f}")

    # Log final accuracy to wandb
    if log_wandb:
        wandb.log({
            'test_accuracy': accuracy,
            'scale_factor': scale_factor,
            'nan_detected': False
        })

    return accuracy, False

# PHASE 2: RL ENVIRONMENT AND AGENT
# ---------------------------------

# RL Environment
class ScaleEnv(gym.Env):
    def __init__(self, min_scale=0.1, max_scale=2.0, num_epochs=5):
        super(ScaleEnv, self).__init__()
        self.min_scale = min_scale
        self.max_scale = max_scale
        self.num_epochs = num_epochs

        # Define action and observation spaces
        self.action_space = spaces.Box(
            low=self.min_scale,
            high=self.max_scale,
            shape=(1,),
            dtype=np.float32
        )

        # Simple observation space - just the previous scale and reward
```

```
        self.observation_space = spaces.Box(
            low=np.array([self.min_scale, -2.0]),
            high=np.array([self.max_scale, 1.0]),
            dtype=np.float32
        )

        self.current_scale = None
        self.current_reward = None

    def reset(self, seed=None):
        super().reset(seed=seed)
        self.current_scale = np.random.uniform(self.min_scale, self.max_scale)
        self.current_reward = 0.0
        return np.array([self.current_scale, self.current_reward]), {}

    def step(self, action):
        # Clip action to ensure it's within bounds
        scale_factor = np.clip(action[0], self.min_scale, self.max_scale)

        # Train and evaluate the network with this scale factor
        accuracy, nan_detected = train_and_evaluate(
            scale_factor=scale_factor,
            num_epochs=self.num_epochs,
            log_wandb=True
        )

        # Compute reward
        if nan_detected:
            reward = -2.0  # Large penalty for NaN
        else:
            reward = accuracy * 10  # Scale up accuracy to have more meaningful gradients

        # Update current state
        self.current_scale = scale_factor
        self.current_reward = reward

        # Always terminate after one step
        done = True
        info = {
            'scale_factor': scale_factor,
            'accuracy': accuracy if not nan_detected else 0.0,
            'nan_detected': nan_detected
        }

        return np.array([self.current_scale, self.current_reward]), reward, done, False, info

# REINFORCE Agent
class ReinforceAgent:
    def __init__(self, state_dim, action_dim, min_scale, max_scale, lr=0.001, gamma=0.99):
        self.gamma = gamma
        self.min_scale = min_scale
        self.max_scale = max_scale
        self.action_range = max_scale - min_scale
        self.action_dim = action_dim
```

```python

        # Policy network: 2-layer MLP outputting mean and log_std
        self.policy = nn.Sequential(
            nn.Linear(state_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, action_dim * 2)  # Output mean and log_std
        ).to(device)

        self.optimizer = optim.Adam(self.policy.parameters(), lr=lr)

        # Save rewards and actions for training
        self.saved_log_probs = []
        self.rewards = []

    def select_action(self, state):
        state = torch.FloatTensor(state).to(device)

        # Forward pass through the policy network
        output = self.policy(state)

        # Split the output into mean and log_std
        mean, log_std = output[:self.action_dim], output[self.action_dim:]

        # Check for NaN values and handle them
        if torch.isnan(mean).any() or torch.isnan(log_std).any():
            print("Warning: NaN detected in policy output. Using fallback action.")
            return np.array([0.5 * (self.min_scale + self.max_scale)])  # Default to middle of range

        # Ensure log_std is not too small for numerical stability
        log_std = torch.clamp(log_std, min=-20, max=2)

        # Create normal distribution
        std = log_std.exp()
        dist = torch.distributions.Normal(mean, std)

        try:
            # Sample action from the distribution
            action = dist.sample()

            # Scale the action to our desired range
            scaled_action = torch.sigmoid(action)  # First to [0, 1]
            scaled_action = scaled_action * self.action_range + self.min_scale  # Then to [min_scale, max_scale]

            # Save log probability for training
            log_prob = dist.log_prob(action)
            self.saved_log_probs.append(log_prob)

            return scaled_action.detach().cpu().numpy()
        except ValueError as e:
            print(f"Error sampling from distribution: {e}")
            print(f"Mean: {mean}, Std: {std}")
            # Return a fallback action
```

```
            return np.array([0.5 * (self.min_scale + self.max_scale)])  # Default to middle of range

    def update(self):
        # Check if we have any rewards to update with
        if len(self.rewards) == 0:
            print("No rewards to update with.")
            return 0.0

        # Convert rewards to returns (discounted cumulative future reward)
        returns = deque()
        R = 0
        for r in reversed(self.rewards):
            R = r + self.gamma * R
            returns.appendleft(R)

        # Convert to tensor
        returns = torch.tensor(returns, device=device)

        # Normalize returns if we have more than 1 element
        if len(returns) > 1:
            try:
                # Use a small epsilon for numerical stability
                returns = (returns - returns.mean()) / (returns.std() + 1e-8)
            except RuntimeError as e:
                print(f"Error normalizing returns: {e}")
                # Don't normalize if there's an error

        # Check if we have any saved log_probs to update with
        if len(self.saved_log_probs) == 0 or len(self.saved_log_probs) != len(returns):
            print("Mismatch between saved log probs and returns.")
            self.saved_log_probs = []
            self.rewards = []
            return 0.0

        # Compute loss
        policy_loss = []
        for log_prob, R in zip(self.saved_log_probs, returns):
            policy_loss.append(-log_prob * R)

        # Check if we have any policy loss to update with
        if len(policy_loss) == 0:
            print("No policy loss to update with.")
            self.saved_log_probs = []
            self.rewards = []
            return 0.0

        policy_loss = torch.cat(policy_loss).sum()

        # Check for NaN in policy loss
        if torch.isnan(policy_loss).any():
            print("NaN detected in policy loss. Skipping update.")
            self.saved_log_probs = []
            self.rewards = []
            return 0.0

```

```
        # Update policy
        self.optimizer.zero_grad()
        policy_loss.backward()

        # Gradient clipping to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(self.policy.parameters(), max_norm=1.0)

        self.optimizer.step()

        # Clear saved rewards and log_probs
        self.saved_log_probs = []
        self.rewards = []

        return policy_loss.item()

# PHASE 3: RL TRAINING AND ANALYSIS
# --------------------------------

def run_rl_training(num_episodes=50, min_scale=0.3, max_scale=1.7, num_epochs=5):
    """Run the RL training loop."""
    # Initialize wandb for tracking
    wandb.init(
        project="rl-nn-initialization",
        name=f"scale_search_{time.strftime('%Y%m%d_%H%M%S')}",
        config={
            "num_episodes": num_episodes,
            "min_scale": min_scale,
            "max_scale": max_scale,
            "num_epochs": num_epochs,
            "device": device.type
        }
    )

    # Create environment
    env = ScaleEnv(min_scale=min_scale, max_scale=max_scale,
    ↪  num_epochs=num_epochs)

    # Get state and action dimensions
    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.shape[0]

    # Create agent
    agent = ReinforceAgent(state_dim, action_dim, min_scale, max_scale)

    # Lists to track progress
    all_rewards = []
    all_scales = []
    all_accuracies = []
    all_nan_flags = []

    # Training loop
    for episode in tqdm(range(num_episodes), desc="RL Training"):
        # Reset environment
        state, _ = env.reset()

        # Select action
        action = agent.select_action(state)
```

```

        # Take action in environment
        next_state, reward, done, _, info = env.step(action)

        # Store reward
        agent.rewards.append(reward)

        # Track metrics
        all_rewards.append(reward)
        all_scales.append(info['scale_factor'])
        all_accuracies.append(info['accuracy'])
        all_nan_flags.append(info['nan_detected'])

        if episode % 5 == 0:
            # Print progress
            print(f"Episode {episode}, Scale: {info['scale_factor']:.4f}, "
                  f"Reward: {reward:.4f}, NaN: {info['nan_detected']}")

        # Log to wandb
        wandb.log({
            'episode': episode,
            'scale_factor': info['scale_factor'],
            'reward': reward,
            'accuracy': info['accuracy'],
            'nan_detected': info['nan_detected']
        })

        # Update the policy every episode
        if done:
            policy_loss = agent.update()
            wandb.log({'episode': episode, 'policy_loss': policy_loss})

    # Run baseline with standard He initialization (s=1.0)
    print("\nEvaluating baseline (s=1.0)...")
    baseline_accuracy, baseline_nan = train_and_evaluate(scale_factor=1.0, num_epochs=num_epochs, log_wandb=True)
    print(f"Baseline (s=1.0): Accuracy = {baseline_accuracy:.4f}, NaN = {baseline_nan}")

    # Find best non-NaN scale factor
    non_nan_indices = [i for i, nan in enumerate(all_nan_flags) if not nan]
    if non_nan_indices:
        non_nan_rewards = [all_rewards[i] for i in non_nan_indices]
        non_nan_scales = [all_scales[i] for i in non_nan_indices]
        non_nan_accuracies = [all_accuracies[i] for i in non_nan_indices]

        best_idx = np.argmax(non_nan_rewards)
        best_scale = non_nan_scales[best_idx]
        best_accuracy = non_nan_accuracies[best_idx]
        print(f"Best scale factor: {best_scale:.4f}, Accuracy: {best_accuracy:.4f}")

        wandb.log({
            'best_scale': best_scale,
            'best_accuracy': best_accuracy,
            'baseline_accuracy': baseline_accuracy
        })
```

```python
    else:
        print("All runs resulted in NaN. Consider adjusting the scale range.")

    return all_scales, all_rewards, all_accuracies, all_nan_flags, baseline_accuracy

# PHASE 4: ANALYSIS AND VISUALIZATION
# ----------------------------------

def analyze_results(scales, rewards, accuracies, nan_flags, baseline_accuracy):
    """Analyze and visualize the results."""
    # Create a figure with two subplots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

    # Plot 1: Reward vs Scale Factor
    ax1.scatter(scales, rewards, alpha=0.6, c=['red' if nan else 'blue' for nan in nan_flags])
    ax1.axvline(x=1.0, color='green', linestyle='--', label='Baseline He Init (s=1.0)')
    ax1.set_xlabel('Scale Factor')
    ax1.set_ylabel('Reward')
    ax1.set_title('Reward vs Scale Factor')
    ax1.grid(True)
    ax1.legend(['NaN Detected' if nan else 'Valid Run' for nan in [True, False]] + ['Baseline He Init (s=1.0)'])

    # Plot 2: Training Progress (Reward over Episodes)
    non_nan_indices = [i for i, nan in enumerate(nan_flags) if not nan]
    if non_nan_indices:
        non_nan_rewards = [rewards[i] for i in non_nan_indices]
        non_nan_scales = [scales[i] for i in non_nan_indices]

        # Add polynomial fit to help visualize trends
        if len(non_nan_scales) > 2:
            try:
                z = np.polyfit(non_nan_scales, non_nan_rewards, 2)
                p = np.poly1d(z)

                # Create smoothed line
                x_poly = np.linspace(min(non_nan_scales), max(non_nan_scales), 100)
                y_poly = p(x_poly)

                ax2.plot(x_poly, y_poly, 'r--', label='Trend')

                # Find the peak of the polynomial
                # For a quadratic, the peak is at -b/(2a)
                if z[0] < 0:  # Check if it's a convex parabola (has a maximum)
                    peak_x = -z[1] / (2 * z[0])
                    if min(non_nan_scales) <= peak_x <= max(non_nan_scales):
                        peak_y = p(peak_x)
                        ax2.scatter([peak_x], [peak_y], c='gold', s=100, zorder=5,
                                    label=f'Predicted Optimal: s{peak_x:.3f}')
            except:
                print("Could not fit polynomial to data")

```

```python
    # Add an artificial "episode" axis to show progression
    episode_numbers = list(range(len(rewards)))
    ax2.scatter(episode_numbers, rewards, alpha=0.6, c=['red' if nan else 'blue' for nan in nan_flags])
    ax2.axhline(y=baseline_accuracy*10, color='green', linestyle='--', label=f'Baseline (s=1.0): {baseline_accuracy:.4f}')
    ax2.set_xlabel('Episode')
    ax2.set_ylabel('Reward')
    ax2.set_title('Reward over Episodes')
    ax2.grid(True)
    ax2.legend()

    plt.tight_layout()
    plt.savefig('rl_init_results.png')
    plt.show()

    # Log chart to wandb
    wandb.log({"results_chart": wandb.Image(fig)})

    # Compute statistics
    non_nan_indices = [i for i, nan in enumerate(nan_flags) if not nan]
    if non_nan_indices:
        best_idx = np.argmax([rewards[i] for i in non_nan_indices])
        best_scale = scales[non_nan_indices[best_idx]]
        best_accuracy = accuracies[non_nan_indices[best_idx]]
        best_reward = rewards[non_nan_indices[best_idx]]

        print("\n=== RESULTS SUMMARY ===")
        print(f"Baseline (s=1.0): Accuracy = {baseline_accuracy:.4f}")
        print(f"Best found scale: s = {best_scale:.4f}, Accuracy = {best_accuracy:.4f}")
        print(f"Improvement over baseline: {(best_accuracy - baseline_accuracy) * 100:.4f}%")

        # Count the number of NaN runs
        nan_count = sum(nan_flags)
        print(f"Total runs: {len(rewards)}, NaN runs: {nan_count} ({nan_count/len(rewards)*100:.1f}%)")

        return best_scale, best_accuracy
    else:
        print("All runs resulted in NaN. Consider adjusting the scale range.")
        return None, None

# Main execution function
def main():
    print("Starting RL Neural Network Initialization Optimization")

    # Run with shorter training to save time
    print("\nPhase 1-3: Running RL training...")

    try:
        # Use narrower scale range to reduce NaNs but with more episodes and epochs
        # Reduced number of episodes and epochs for MacBook to run faster
        scales, rewards, accuracies, nan_flags, baseline_accuracy = run_rl_training(
```

```
            num_episodes=200,     # Reduced from 200 to 50 for faster execution
            min_scale=0.1,       # Increased from 0.1 to avoid very small scales that can cause NaNs
            max_scale=1.9,       # Decreased from 2.0 to avoid very large scales that can cause NaNs
            num_epochs=7         # Reduced from 10 to 7 for faster execution
        )

        print("\nPhase 4: Analyzing results...")
        best_scale, best_accuracy = analyze_results(
            scales, rewards, accuracies, nan_flags, baseline_accuracy
        )

        if best_scale is not None:
            # Validate the best scale with a longer training run
            print("\nValidating best scale with longer training...")
            final_accuracy, final_nan = train_and_evaluate(
                scale_factor=best_scale,
                num_epochs=20,  # Reduced from 20 to 10 for faster execution
                log_wandb=True
            )
            baseline_final, _ = train_and_evaluate(
                scale_factor=1.0,
                num_epochs=20,  # Same length for fair comparison
                log_wandb=True
            )

            print(f"\nFinal validation (10 epochs):")
            print(f"Best scale (s={best_scale:.4f}): Accuracy = {final_accuracy:.4f}")
            print(f"Baseline (s=1.0): Accuracy = {baseline_final:.4f}")
            print(f"Difference: {(final_accuracy - baseline_final) * 100:.4f}%")

            # Log final results to wandb
            wandb.log({
                'final_best_scale_accuracy': final_accuracy,
                'final_baseline_accuracy': baseline_final,
                'final_improvement': (final_accuracy - baseline_final) * 100
            })
    except Exception as e:
        print(f"An error occurred during execution: {e}")

    # Close wandb run
    wandb.finish()

if __name__ == "__main__":
    main()
```

Appendix B

PLAGIARISM REPORT

Document Details

Submission ID

trn:oid:::27535:97238589

Submission Date

May 22, 2025, 9:45 PM GMT+5:30

Download Date

May 22, 2025, 9:47 PM GMT+5:30

File Name

Reinforcement_learning_for_weight_initialisation.pdf

File Size

880.9 KB

54 Pages

16,491 Words

92,348 Characters

turnitin Page 1 of 62 - Cover Page Submission ID trn:oid:::27535:97238589

10% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text
- Cited Text
- Small Matches (less than 10 words)

Match Groups

83 Not Cited or Quoted 10%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

0 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

7% Internet sources
4% Publications
9% Submitted works (Student Papers)

Integrity Flags

1 Integrity Flag for Review

Replaced Characters
24 suspect characters on 9 pages
Letters are swapped with similar characters from another alphabet.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Bibliography

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, JMLR Workshop and Conference Proceedings, 2010.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034, 2015.

[3] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[4] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2017.

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

[6] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[8] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 ed., 2010.

[9] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 9, pp. 249–256, 2010.

[10] V. Nair and G. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 807–814, 2010.

[11] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pp. Volume 28, Issue 1, 2013.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

[13] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *International Conference on Learning Representations (ICLR)*, 2016.

[14] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[15] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," *arXiv preprint arXiv:1312.6120*, 2013.

[16] D. Mishkin and J. Matas, "All you need is a good init: Initializing convolutional neural networks for fast and stable training," *arXiv preprint arXiv:1511.06422*, 2016.

[17] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," *arXiv preprint arXiv:1312.6120*.

[18] L. Li, K. Swersky, A. Shahriari, and J. Dean, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, pp. 1–52, 2017.

[19] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *International Conference on Learning Representations (ICLR)*, 2017.

[20] M. Towers *et al.*, "Gymnasium: An api for reinforcement learning," *arXiv preprint arXiv:2308.02633*, 2023.

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

[22] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[23] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* A Bradford Book, The MIT Press, 2nd ed., 2018.

[24] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *unpublished manuscript*, 2009.

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

[26] L. Biewald, "Experiment tracking with weights and biases," 2020.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 8026–8037, 2019.

[28] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[29] J. Towers *et al.*, "Gymnasium: A standard api for reinforcement learning environments," 2023. https://github.com/Farama-Foundation/Gymnasium.

[30] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.