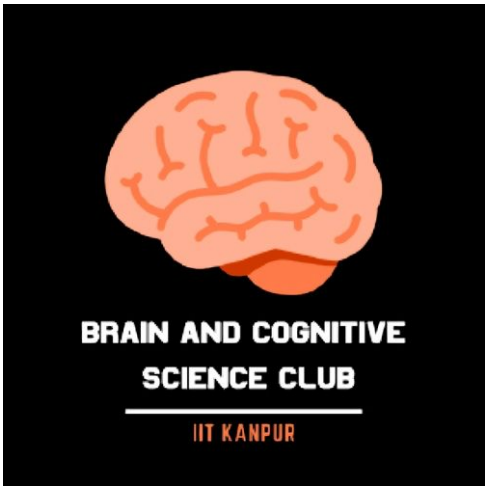


BCS Spikeverse

Brain and Cognitive Science Club, IIT Kanpur

Mentors: Gaurav Rampuria, Kshitiz Tyagi, Saubhagya Pandey

Abstract

Project Overview: This project explores the integration of **Spiking Neural Networks (SNNs)** with **Reinforcement Learning (RL)**, focusing on the **ANN-to-SNN conversion** for game-playing agents.
We start with fundamental RL concepts, implement a **Deep Q-Network (DQN)** for Atari Breakout, and then convert the ANN-based model to an SNN. Thus, we study performance trade-offs, computational efficiency, robustness, and alternative neuron models.

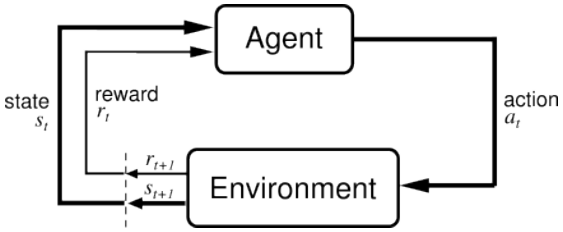
Date: 13 June 2025

Contents

1 Introduction to Reinforcement Learning

Reinforcement learning (RL) is a branch of Machine learning, where an **agent** learns to choose an action from its action space, within a particular **environment** which leads it to **maximize** the cumulative rewards **overtime**.

1.1 Key Components of RL



The standard RL framework consists of:

- **Agent**: The decision maker, which learns from *experience*
- **Environment**: The system with which the agent interacts with in order to recieve **rewards** and next *states* as transitions.
- **State** ($s \in \mathcal{S}$): The state should describe the relevant information we need in order to decide which action to take at timestep t.
- **Action** ($a \in \mathcal{A}$): A choice the agent makes. It transitions the agent to a later state in the environment (apart from its current one)
- **Reward** ($r \in R$): The reward function tells us after every action how much reward the agent got for taking that action.
- **Policy** ($\pi(a|s)$): A mapping required to guide the behaviors. It maps from states to probabilities.

1.2 Markov Decision Process (MDP)

Reinforcement Learning problems are mathematically formulated as **Markov Decision Processes (MDPs)**, defined by the 5-tuple:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$$

- $\mathcal{S}$: Set of possible states
- $\mathcal{A}$: Set of possible actions
- $\mathcal{P}(s'|s,a)$: Probability of transitioning to state s' from state s under action a
- $\mathcal{R}(s,a)$: Expected reward received when taking action a in state s
- γ: Discount factor ($0 \leq \gamma \leq 1$) that balances immediate vs future rewards

If an environment has the **Markov property**, then its one-step dynamics enable us to predict the next state and expected next reward given the current state and action. One can show that, by iterating this equation, one can predict all future states and expected rewards from knowledge only of the current state as well as would be possible given the complete history up to the current time.

1.3 Agent–Environment Interaction

Agent and environment interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent.3.1 The environment also gives rise to rewards, special numerical values that the agent tries to maximize over time. A complete specification of an environment defines a task, one instance of the reinforcement learning problem.

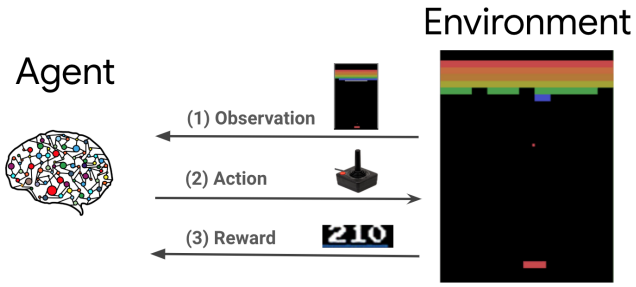


Figure 1: Agent-Environment loop in our current interest, i.e., Atari Breakout setup.

1.4 Q-Learning and the Bellman Equation

Q-learning is a popular reinforcement learning algorithm. The algorithm leverages the (Bellman Equation) to estimate the value of state-action pairs, known as Q-values. These values tell the agent how rewarding it will be to take a certain action in a given state and follow the best policy afterward.

$$Q^*(s,a) = E_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right]$$

This equation reflects the principle that the value of a state-action pair equals the imme-

diate reward plus the discounted value of the best action in the next state. It captures the *bootstrapping* nature of reinforcement learning, where future estimates are used to update current ones.

In practice, Q-learning uses the following update rule after each step:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where:

- $\alpha \in (0,1]$: Learning rate controlling how much new information overrides old knowledge.
- r: The reward received after taking action a in state s
- s': The new state reached after the transition
- γ: Discount factor determining the importance of future rewards

The term inside the brackets is known as the **temporal difference (TD) error**:

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

This error measures the gap between the current estimate and the one-step improved estimate using the Bellman equation. Q-learning iteratively reduces this error, converging to the optimal Q-function under suitable conditions.

1.5 Tabular Q-Learning in Practice

Deep Q-Learning or Deep Q Network (DQN) is an extension of the basic Q-Learning algorithm, which uses deep neural networks to approximate the Q values.

However, even in moderately sized environments, the number of state-action pairs grows exponentially. In Atari Breakout, the state is a 210×160 RGB image—representing millions of possible states. It's computationally impossible to store a table for all such combinations.

Traditional Q-Learning works well for environments with a small and finite number of states, but it struggles with large or continuous state spaces due to the size of the Q-table. Deep Q-Learning overcomes this limitation by replacing the Q-table with a neural network that can approximate the Q-values for every state-action pair.rs like (neural networks), leading to Deep Q-Networks (DQN), which we introduce in the next section.

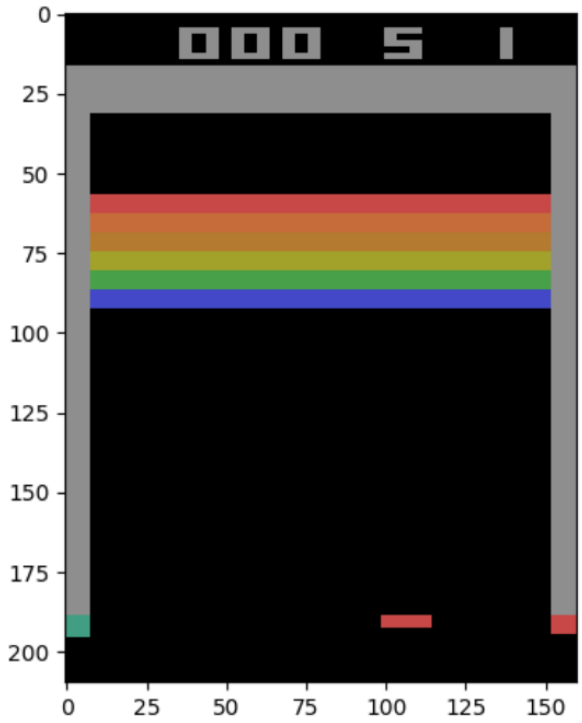


Figure 2: Atari env as 210x160 RGB image, unprocessed.

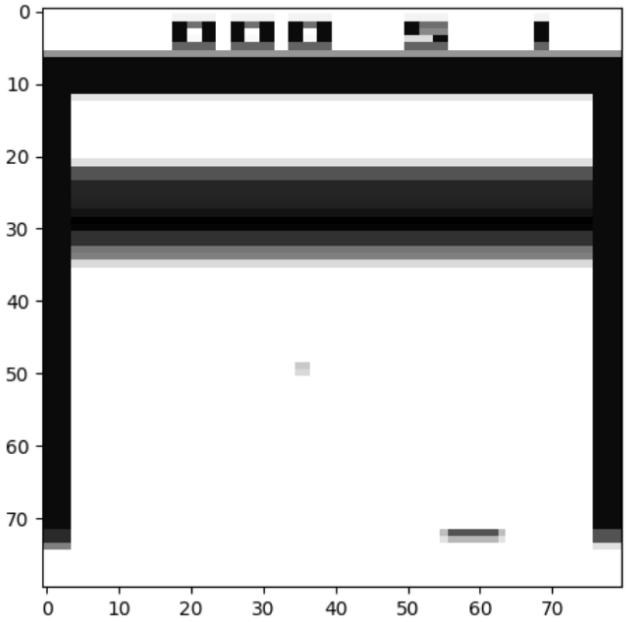


Figure 3: Atari env as 84x84 B/W image, processed.

2 Deep Q-Networks (DQN)

A **Deep Neural Network (DNN)** used to estimate the **Q-values** is called a **deep Q-network (DQN)** and using a DQN for **Approximate Q-learning** is called **Deep**

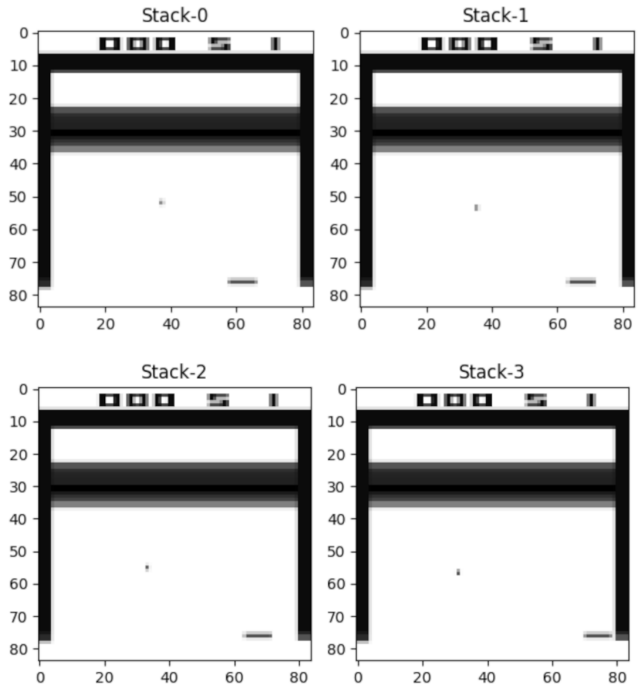


Figure 4: Atari env as 84x84 B/W image, 4 frames stacked to be fed into CNN layers.

Q-Learning. Now, the question is, how do we train a DQN? The approximate Q-value is computed by the DQN for a given **state-action pair (s,a)**.

This approximate Q-value should be as close as possible to the **reward r** collected by the agent after playing an action in a **state s**, plus the **discounted value** of the agent playing optimally from then on. To estimate this future discounted value, we can run the DQN on the next **state s'** and for all possible **actions a'** for that particular state. From the approximate Q-values of all the possible actions for that state s', we pick the **highest Q-value** (agent should play optimally).

Then we discount this Q-value with the **discount factor** γ. This future discounted value estimate is summed with the reward r, giving us the **target Q-value** $y(s, a)$ for the state-action pair (s, a). With the target Q-value we can run this training iteration with any **Gradient Descent Policy**. The **squared error** between the target Q-value and estimate Q-value is minimised. This is the basic Deep Q-Learning algorithm.

After regular intervals, the **weights** of the **online DQN** are copied to the **target DQN**. This dramatically improves the performance of the algorithm. Using a single network here is like a dog chasing its own tail. It is not a good idea to use the same function to update its estimates because this leads to an **'unstable' target function**.

The difference between Q-learning and DQN is that you have replaced an **exact value function** with a **function approximator**. With Q-learning you are updating exactly **one state/action value at each timestep**, whereas with DQN you are updating many, which you understand. The problem this causes is that you can affect the **action values for the very next state** you will be in instead of **guaranteeing them to be stable** as they are in Q-learning.

This happens basically all the time with DQN when using a **standard deep network** (bunch of layers of the same size fully connected). The effect you typically see with this is referred to as **"catastrophic forgetting"** and it can be quite spectacular. If you are doing something like **Cartpole env with this sort of network** and track the **rolling average score over the last 150 games or so**, you will likely see a nice curve up in score, then all of a sudden it **completely dies out**and starts making **awful decisions again** even as your **alpha gets small**. This cycle will continue endlessly regardless of how long you let it run.

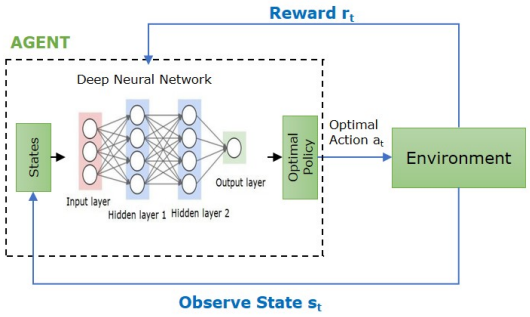


Figure 5: Example of DQN training Environment. This DQN has 2 hidden layers

2.1 Target Networks

Using a stable target network as your error measure is one way of **combating this effect**. Conceptually it's like saying, **"I have an idea of how to play this well, I'm going to try it out for a bit until I find something better"** as opposed to saying **"I'm going to retrain myself how to play this entire game after every move"**.

By giving your network **more time to consider many actions that have taken place recently** instead of updating all the time, it hopefully **finds a more robust model** before you start using it to make actions.

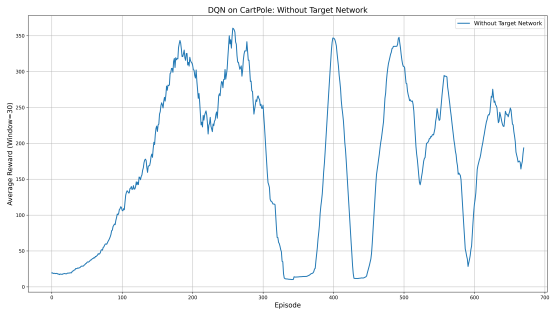


Figure 6: Cartpole Model(Without target network) goes up only till epoch 170 and then it gets garbage

2.2 Experience Replay in DQN

Experience Replay is a foundational technique in Deep Q-Networks (DQN). It involves storing the agent's experiences at each time step - usually represented as a tuple - (s_t, a_t, r_t, s_{t+1}) - in a replay memory or buffer. During training, the agent randomly samples mini-batches of these stored experiences to update the neural network, rather than learning only from the most recent experience. This approach offers several key benefits:

- **Breaks correlation** between sequential experiences, leading to **more stable** and efficient learning.
- **Improves sample efficiency** by allowing the agent to learn from past experiences multiple times.
- **Mitigates ruinous forgetting** by ensuring older experiences are not immediately discarded.

In DQN , experience replay **transforms the learning process to be more like supervised learning,** helping to stabilize and accelerate convergence.

3 Improving Vanilla DQN

3.1 The Problem

Researchers found that **Q-learning** tends to **overestimate the value** of actions it can take. This overestimation happens because the algorithm always chooses the action it believes has the **highest value**—even if that value is based on **noisy or inaccurate information**.

These overestimations aren't just small mistakes—they can lead to the algorithm learning **bad or suboptimal behaviors**, even after a lot of training. This issue becomes worse when the algorithm uses methods like **function approximation** or when the environment itself is **noisy or unpredictable**.

Later it was showed that this overestimation can occur even in very simple settings, and proposed a solution called **Double Q-learning**, which reduces the bias by using **two separate estimates** to make decisions.

3.2 The solution: DDQN

Double DQN (DDQN) addresses this issue by introducing a simple but effective change: it **decouples the action selection and evaluation steps**. Specifically, DDQN uses the **online network** to select the **best next action**, but evaluates the value of that action using a separate **target network**. This separation helps prevent the model from **"trusting its own hype"** when making updates. Since the two networks are not identical and are **updated at different times**, their errors are less likely to reinforce each other, which leads to more **accurate and stable value learning**.

The real strength of DDQN lies in how it balances **optimism and caution**. Because the action is chosen using one network and evaluated using another, there's a much lower chance that both steps will be biased in the same direction. As a result, the **estimated Q-values** are less likely to be systematically too high or too low. This more balanced estimation not only **improves stability during training** but also allows the agent to **learn better decision-making policies** in a wide range of environments, especially where **deep function approximation** is used.

3.3 Prioritized Experience Replay (PER)

PER enhances standard **experience replay** by **sampling experiences with non-uniform probability**. Instead of treating all experiences equally, PER assigns a **priority** to each experience, typically based on the magnitude of its **temporal-difference (TD) error**. Experiences with **higher TD errors, those from which the agent can learn the most,** are sampled more frequently.

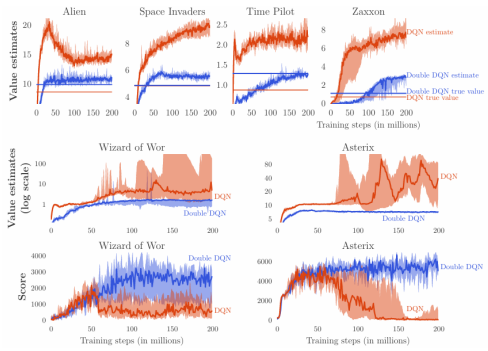


Figure 7: Comparing DQN and Double DQN performance in multiple Atari 2600 games.

3.3.1 Key Concepts

- **Priority Assignment:** The priority p_i of each experience is often set to the absolute TD error $|\delta_i|$, reflecting how **surprising or informative** the experience is.
- **Sampling Probability:** Each experience i is sampled with probability

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where α controls the degree of **prioritization** ($\alpha = 0$ reduces to uniform sampling).

- **Importance Sampling Weights:** To correct for the bias introduced by non-uniform sampling, each sampled experience is weighted by

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

where β **anneals from an initial value to 1** over training, ensuring **unbiased updates** as learning progresses.

- **Diversity and Bias: Stochastic prioritization** ensures even low-priority experiences have a non-zero chance of being sampled, preserving **diversity** and avoiding **overfitting**.

3.3.2 Integration into Network Architecture

DQN: PER improves DQN by focusing learning on the **most informative experiences**, leading to **faster and more robust convergence**, especially in environments with **sparse or unbalanced rewards**.

Double-DQN: PER is compatible with Double-DQN, where it further helps **reduce overestimation bias** by ensuring **critical transitions are revisited more often**, and **importance sampling weights** become especially important to **maintain unbiased value estimates**.

In short, PER uses **TD error as a proxy for learning potential** and corrects for **sampling bias** with **importance sampling weights**.

The following plot shows:: *Difference in normalized score (the gap between random and human is 100 percent) on 57 games with human starts, comparing Double DQN with and without prioritized replay (rank-based variant in red, proportional in blue), showing substantial improvements in most games.*

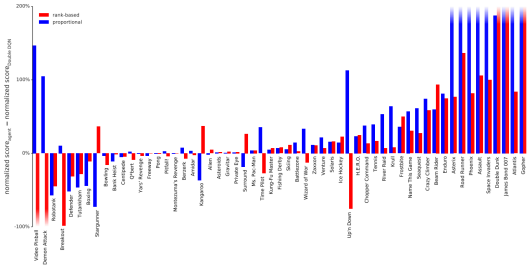


Figure 8: Double DQN with and without PER in 57 Games with human starts

4 Rainbow: A Unified DQN

While **Double DQN (DDQN)** and **Prioritized Experience Replay (PER)**—discussed in the previous section—address **overestimation bias** and **sample efficiency**, Rainbow DQN further strengthens vanilla DQN with four additional innovations. Together, these six improvements make Rainbow DQN a highly robust and generalizable deep reinforcement learning agent.

The remaining four components of Rainbow DQN are:

4.1 Dueling Network Architecture

In Atari games like **Breakout**, many states (e.g., ball mid-air) don't vary much in value across different actions. The **Dueling DQN** addresses this by decomposing Q-values into:

- **State value** $V(s)$ — how good the frame stack is overall
- **Advantage** $A(s, a)$ — how much better one paddle move is vs others

They are combined as:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

This improves learning when many actions are equally valuable. It is particularly useful in 84×84 stacked-frame Breakout where the ball's position may not change Q-values much across actions. The core intuition behind the Dueling Network Architecture is that in many states, estimating the value of each action is not equally important.

In Breakout, consider a state where the ball is far from the paddle—whether the paddle moves left, right, or stays still often doesn't affect the outcome immediately. In such cases, it's more useful to estimate the overall value of the state rather than the value of each individual action.

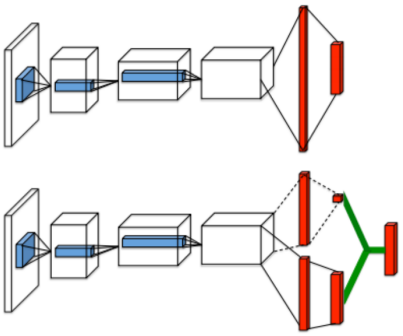


Figure 9: Traditional (top) vs. Dueling (bottom) Q-network. The bottom architecture separates and recombines value and advantage streams.

4.2 N-Step Learning

N-step temporal difference (TD) learning comes from the idea of balancing between shallow updates (as in TD methods) and deep backups (as in Monte Carlo methods). Monte Carlo methods execute entire episodes and then backpropagate the reward, whereas TD methods only look at the reward in the next step. **N-step methods instead look n steps ahead** before updating the Q-function, and then estimate the remainder.

We will look at n-step reinforcement learning, in which n **is the parameter that determines the number of steps** to look ahead before updating the Q-function. So for $n = 1$, this is just "normal" TD learning such as Q-learning or SARSA. When $n = 2$, the algorithm looks one step beyond the immediate reward; $n = 3$ looks two steps beyond, etc.

At an intuitive level, it is quite straightforward: **instead of updating our Q-function based on the immediate reward**, we update it based on the last n rewards received, plus the discounted future rewards from n states ahead. The **n-step return** is defined as:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n})$$

The n-step return $G_t^{(n)}$ consists of the sum of discounted rewards over n steps, plus a bootstrapped estimate of the value $V(s_{t+n})$ at the future state. This allows the agent to assign credit over multiple steps instead of just one.

The basic idea is that we do not update the Q-value immediately after executing an action: we wait n steps and update it based on the n-step return. If $t + n \geq T$ (where T is terminal), then we just use the full reward and drop the bootstrapped estimate.

Atari Breakout is a **sparse reward environments**, here this technique is more useful. Agent would struggle initially to find high rewarding states. But by **propagating rewards across multiple steps**, we'll achieve more informative updates and sped up learning.

Compared to 1-step learning, **n-step learning spreads the states influencing the rewards across timeline**, thus improving credit assignment.In breaout env, reaching a reward requires coordinated paddle movements and ball timing.

4.3 Noisy Networks for Exploration

Meire Fortunato introduced **NoisyNet** in **2018 Paper**, it serves a nice and efficient alternative to the **Epsilon-Greedy** exploration strategy in **DQN**. It improves the efficiency by **injecting noise into the parameters** of the output layer in a neural network. In NoisyNet, the agent explores whenever there is **noise in the sampled output** of the neural network. This is accomplished by replacing traditional **dense linear layers** with **noisy dense layers**. Crucially, the **variance of the noise** is not fixed; it is **learned** alongside the other parameters of the network, via gradients from the **reinforcement loss function**.

The **noisy linear layer** is computed using factorised Gaussian noise. The noise parameters, denoted as ϵ, are sampled separately and combined with trainable parameters. All parameters in the layer are trainable except for the epsilon terms, which are generated using

factorised Gaussian noise. The trainable parameters include μ and σ, where the agent **learns the weights of the sigma terms** over time.

Eventually, the agent may reduce the magnitude of σ, effectively learning to **attenuate noise** as needed. Unlike epsilon-greedy methods that use a **predefined exploration schedule**, NoisyNet enables the agent to **tune the level of exploration dynamically** in response to the learning environment.

In their original work [**?**], **Meire et al.** reported achieving **sub-human to super-human performance** across most Atari games when NoisyNet was applied as the exploration strategy to **DQN**, **A3C**, and **Dueling DQN** agents. However, it was observed that **NoisyNet A3C performed poorly on the Breakout game**, and therefore was not used in subsequent A3C experiments.

The network design for our experiments using **NoisyNet DQN** is shown in Figure

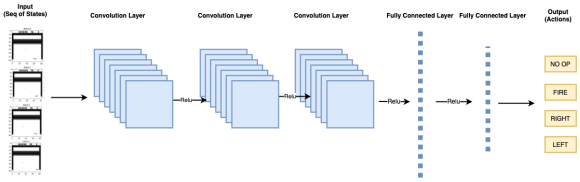


Figure 10: Architecture of Normal DQN; ReLU nonlinear units. 4-stacked frames are fed to DQN.

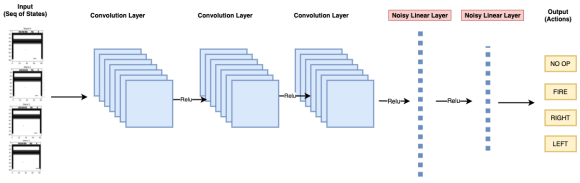


Figure 11: Network architecture: after adding noisy DQN to add noise to parameters.

4.4 Distributional Q-learning

Traditional DQN estimates the **expected return** $Q(s, a)$ as a single scalar. **Distributional Q-learning**, on the other hand, models the **entire distribution** over possible returns $Z(s, a)$.

This approach captures the **uncertainty and variability** in returns, leading to better risk-sensitive decisions. The agent no longer learns just an average outcome, but rather a **probability distribution over returns**, which helps in:

- **Stabilizing learning**,
- **Improving exploration**, and
- **Capturing multi-modal reward patterns**.

The Categorical DQN (C51) is a well-known implementation of this idea, where the return distribution is approximated using a fixed set of atoms and categorical probabilities.

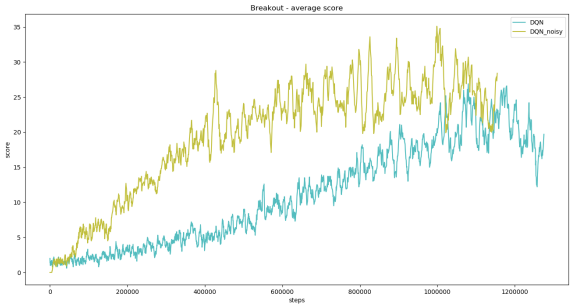


Figure 12: Average score curve for Breakout env, with and without noisy networks

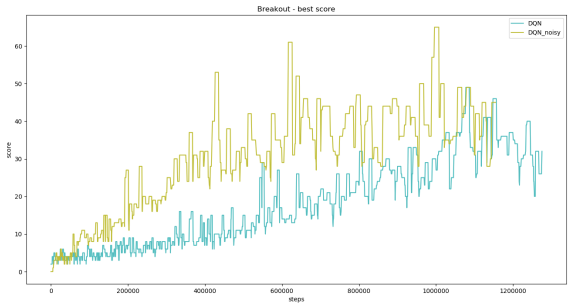


Figure 13: Best score curve of breakout env, with and without noisy networks

In summary, these four additions—**Dueling Networks**, **Multi-step Learning**, **Noisy Networks**, and **Distributional Q-learning**—work in synergy with **DDQN** and **PER** to create the full **Rainbow DQN** agent. Together, they address critical challenges like **bias, variance, exploration, sample efficiency**, and **reward propagation**, resulting in state-of-the-art performance on a wide range of benchmark environments like Atari 2600.

5 Spiking Neural Networks

We begin this section by addressing the problem visible in Figure **??**: the cost of computing state-of-the-art AI models has become

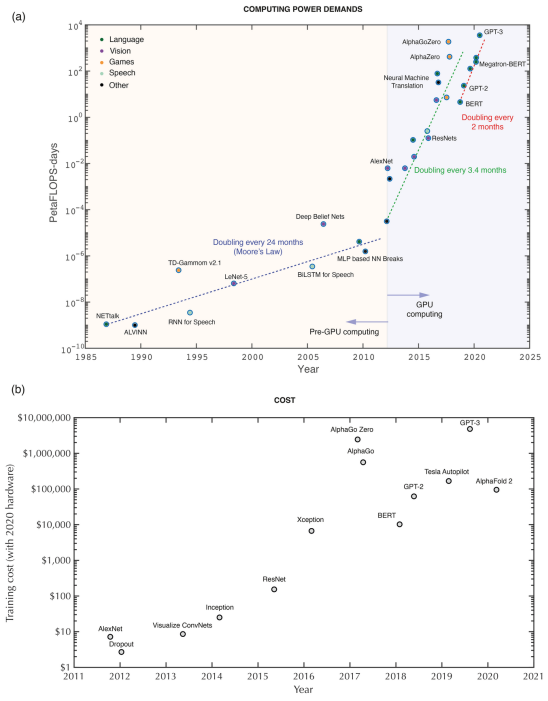


Figure 14: (a) The introduction of GPUs accelerated computational progress significantly. (b) Cost for training state-of-the-art models remains very high (e.g., GPT-3 costed more than 1M dollars).

tremendously large. Training such large models requires GPUs running continuously. Even after training, using ANN-based models requires huge maintenance and consumes a lot of power, needing substantial cooling resources, which negatively impacts our environment.

Neuromorphic systems, designed to mimic the brain's event-driven processing, offer a fundamentally different computational paradigm. Unlike traditional CPUs and GPUs that perform dense matrix operations and require constant memory transfers (often leading to the Von Neumann Bottleneck), neuromorphic hardware works with **sparse spikes**. This event-driven nature activates computation only when spikes occur, dramatically reducing energy usage compared to previous generations of computing.

5.1 Transition to SNNs

Spiking Neural Networks (SNNs) are the algorithmic counterparts of neuromorphic hardware. As the third generation of Artificial Neural Networks (ANNs), they more closely mirror the functionality of the mammalian brain. Their computational units, **spiking neurons**, are characterized by Ordinary Differential Equations (ODEs) and allow for asynchronous, dynamic communication via spikes. This makes SNNs promising for deep networks used in Reinforcement Learning (RL) tasks.

In our case, we took our previously trained ANN for Atari Breakout and converted it into a **spiking neural network** for faster and more energy-efficient prediction of suitable actions.

We demonstrate that ReLU-based NNs trained with RL algorithms can be converted to SNNs without degrading performance on the RL task. Moreover, the converted SNN is more robust to input perturbations than the original ANN. Finally, we show that a full-sized Deep Q-Network (DQN) can be converted into an SNN while maintaining its better-than-human performance, paving the way for research in robustness and RL using SNNs.

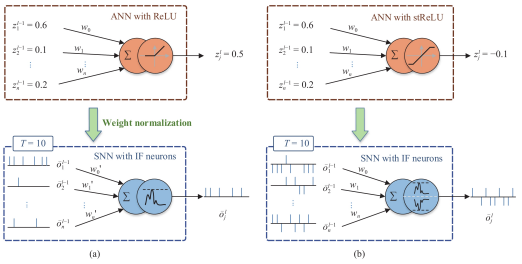


Figure 15: A glimpse of the weight-transfer method. Improving accuracy remains an active research area.

For weight transfer, we scaled the weights by a factor of 10 at each successive layer, as spikes become increasingly sparse deeper in the network.

The next sections will explore the working of SNNs in greater detail and their inspiration from **neurobiology**.

5.2 Leaky Integrate-and-Fire (LIF) Model

The Leaky Integrate-and-Fire (LIF) model simulates the evolution of a neuron's membrane potential over time. Like an artificial neuron, it integrates the weighted sum of inputs. However, instead of directly applying an activation function, it accumulates the potential over time, introducing a decay or *leakage*—akin to the behavior of an RC (resistor-capacitor) circuit.

If the integrated membrane potential exceeds a defined threshold, the neuron emits a discrete **spike**. This spike is treated as a binary event

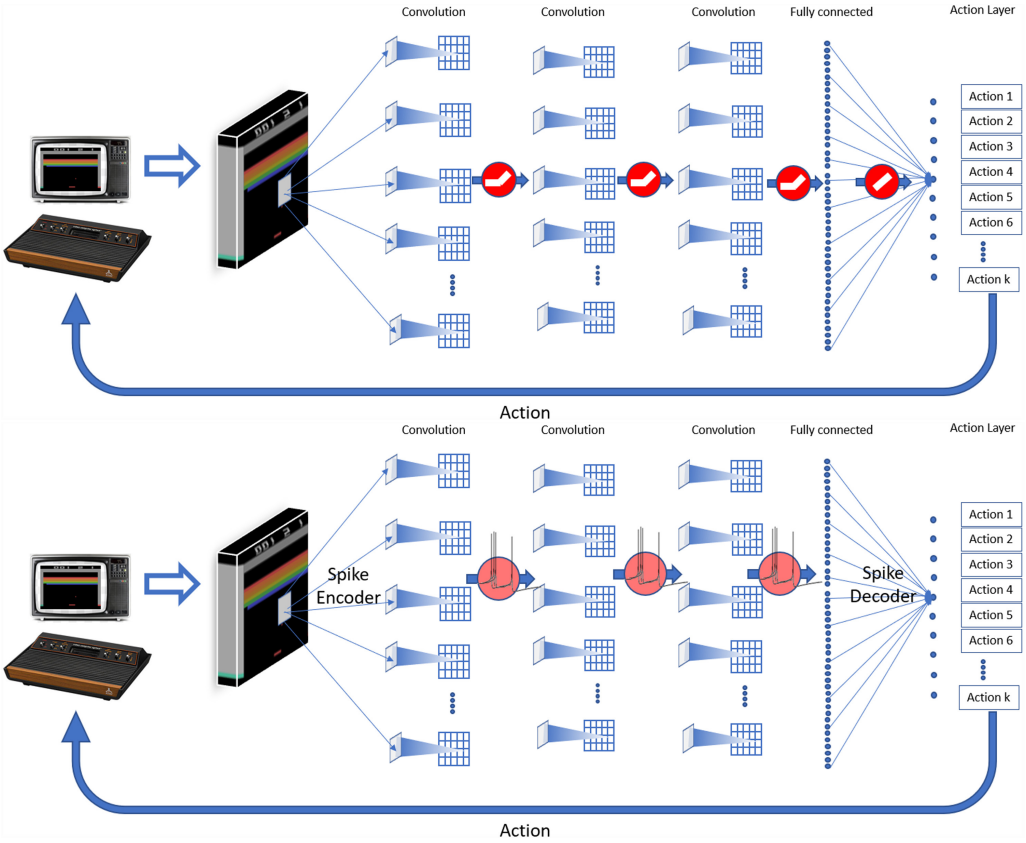


Figure 16: (Top) Previously trained ANN with ReLU activation on the Atari Breakout V4 environment by OpenAI Gym. (Bottom) The same ANN converted to a Spiking Neural Network (SNN) architecture using weight transfer.

— information is carried not in the shape or magnitude of the spike, but in its *timing* or *firing rate*.

Key Properties of the LIF Neuron:

- **Integration:** Incoming inputs are summed and integrated over time.
- **Leakage:** The potential decays exponentially toward a resting value.
- **Firing Threshold:** When the membrane potential crosses the threshold, a spike is emitted.
- **Reset:** After firing, the potential is reset to a resting state.

We use the following notations to describe the dynamics:

- $v(t)$: Membrane potential at time t
- v_{rest}: Resting membrane potential
- v_{thresh}: Firing threshold
- τ: Time constant (membrane decay rate)
- W_i: Synaptic weight of input i
- $Input_i$: Input spike to synapse i

The LIF model is governed by the following differential equation:

$$\tau \frac{dv(t)}{dt} = -(v(t) - v_{rest}) + \sum_{i=1}^{n} W_i \cdot Input_i$$

When $v(t) \geq v_{thresh}$, the neuron emits a spike and $v(t)$ is reset to its resting value.

5.3 Spike-Timing Dependent Plasticity (STDP)

Spike-Timing Dependent Plasticity (STDP) is a biologically inspired learning rule based on the relative timing of spikes between pre-synaptic and post-synaptic neurons.

If the pre-synaptic neuron's spike occurs **before** the post-synaptic neuron, their relationship is considered *causal*, and the synaptic weight is strengthened. Conversely, if the pre-synaptic spike occurs **after** the post-synaptic neuron fires, the relationship is *acausal*, and the synaptic weight is weakened.

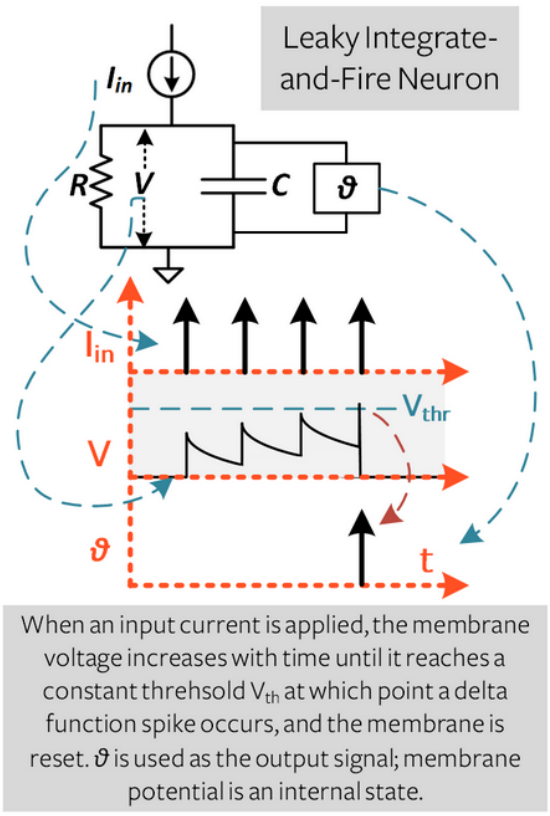


Figure 17: Circuit-level visualization of a Leaky Integrate-and-Fire (LIF) neuron. The membrane potential V integrates incoming current I_{in} through a resistor-capacitor pair. When V exceeds V_{thr}, a spike ϑ is emitted and the membrane is reset.

This gives rise to:

- **Long-Term Potentiation (LTP):** Increase in synaptic strength when $\Delta t = t_{post} - t_{pre} < 0$
- **Long-Term Depression (LTD):** Decrease in synaptic strength when $\Delta t > 0$

Mathematically, the weight change Δw is often modeled using exponential decay functions:

$$\Delta w = \{ A_+ \cdot e^{\Delta t/\tau_+}, if \Delta t < 0$$

-A$_-$ $\cdot e^{-\Delta t/\tau_-}, if \Delta t > 0$

where:

- $\Delta t = t_{post} - t_{pre}$ is the spike time difference
- A_+, A_- are scaling factors for potentiation and depression
- τ_+, τ_- are time constants for LTP and LTD windows

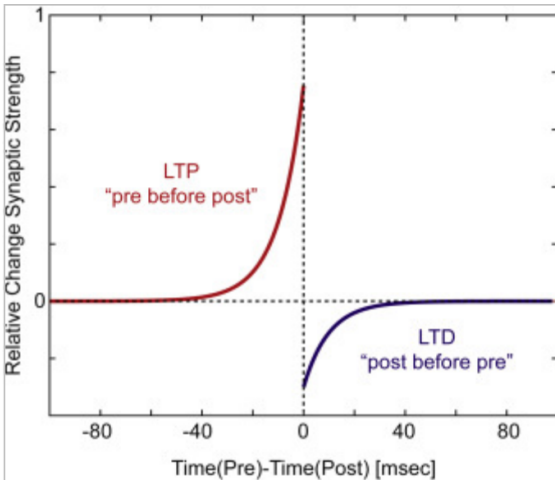


Figure 18: Weight change curve in STDP based on the spike timing difference Δt. LTP occurs when $\Delta t < 0$, and LTD occurs when $\Delta t > 0$.

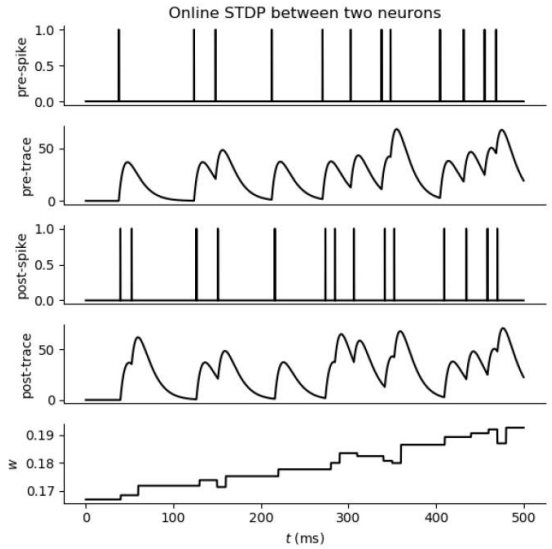


Figure 19: We can see output spikes occur only when input spike train in condensed (More details can be inferred from membrane potential plot)

6 Motivation for Surrogates

Training traditional neural networks involves the use of backpropagation, which relies on the computation of gradients. However, in SNNs, the spike function is a non-differentiable thresholding function, preventing direct use of gradient-based optimization.

6.1 The Problem with the Spike Function

The spike generation mechanism in spiking neurons is typically defined as:

$$S(t) = H(u(t) - \theta)$$

H is the Heavside function. where $u(t)$ is the membrane potential at time t, and θ is the threshold potential.

While this function accurately captures the binary nature of spike emission, it is inherently non-differentiable and has a gradient of zero almost everywhere. As a result, standard gradient-based optimization techniques such as backpropagation cannot be directly applied. This limitation gives rise to the use of **surrogate gradients**, which approximate the gradient of the spike function with a smooth, differentiable alternative during the backward pass.

6.2 The Surrogate Gradient Approach

To enable learning in SNNs, the surrogate gradient approach proposes using the original spike function during the forward pass, but a smooth, differentiable surrogate function during the backward pass. Formally, if $H(x)$ is the Heaviside step function, then:

$$H(x) \to \tilde{H}(x) \quad (forbackwardpass)$$

$$\frac{dH}{dx} \approx \frac{d\tilde{H}}{dx}$$

6.3 Common Surrogate Derivatives

Several surrogate functions are used to approximate the derivative of the spike function:

- **Fast Sigmoid Derivative:**

$$\frac{1}{(1+|x|)^2}$$

- **Exponential Function:**

$$e^{-x^2}$$

- **Piece-wise Linear Function:**

$$\frac{dS}{dx} \approx \max(0, 1 - |x|)$$

- **Cosh-based Function:**

$$\frac{1}{\cosh^2(x)}$$

These approximations help pass non-zero gradients during training and allow SNNs to be trained similarly to traditional neural networks.

6.4 Example and Explanation

The following figure illustrates how surrogate activations and their derivatives are used to approximate the original non-differentiable spike function.

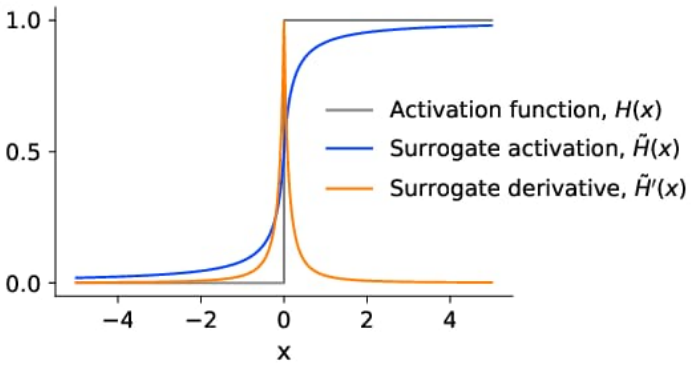


Figure 20: The step function $H(x)$, its smooth approximation $\tilde{H}(x)$, and its surrogate derivative $\tilde{H}'(x)$.

As seen in Figure 20, the actual activation function (step) is approximated by a smooth sigmoid-shaped curve. Its derivative, while non-zero around the threshold, vanishes away from it — mimicking the locality of spike generation.

6.5 Comparison of Surrogate Gradients

The next figure shows a comparison of various surrogate derivative shapes. Each has different sharpness and support, influencing how gradients are propagated.

- The **fast sigmoid** provides a soft gradient with a gentle slope.
- The **piecewise linear** has a clear cutoff and is computationally efficient.
- The **cosh-based** is sharper and more localized.
- The **exponential** is widely used for its biological plausibility.

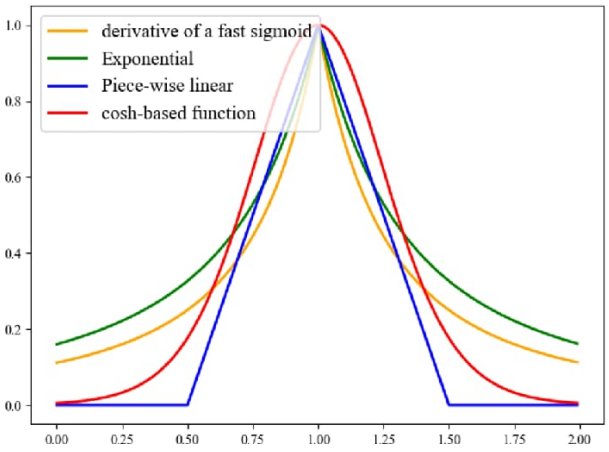


Figure 21: Comparison of surrogate gradient approximations: fast sigmoid, exponential, piecewise-linear, and cosh-based functions.

6.6 Problems in Surrogate Gradient Learning

Although surrogate gradient learning has made it possible to train deep-spiking neural networks using backpropagation, we did not use it because of the following reasons.

- They are largely unstable in RL settings, because the reward is very delayed. This makes backpropagation and spikes inherently unstable.
- As spikes are binary events, gradient signals often become very sparse, which results in extremely slow convergence. This makes it impractical to use them for game environments such as Atari.

7 ANN-to-SNN Conversion

Given the computational limitations and instability of direct SNN training methods such as surrogate gradient descent and STDP, we adopted an efficient alternative: training a Deep Q-Network (DQN)-based ANN for Atari Breakout, and converting it into an SNN through direct weight transfer. This method enabled us to retain the performance of the ANN while gaining the inference-time efficiency benefits of SNNs.

7.1 Why Surrogates Were Discarded

We experimented with training SNNs using surrogate gradient-based Double DQN and prioritized experience replay. However, the results were unsatisfactory:

- Training was extremely slow, even with limited time steps (e.g., 25 steps took ∼15 minutes for 50 episodes).
- Performance did not improve significantly despite long runtimes.
- Hyperparameter tuning was unstable and sensitive.

Due to these challenges, surrogate-based training was not viable for the Breakout environment, especially under real-world constraints.

7.2 Analysis of MNIST with SNN

We tried converting a simple ANN, like MNIST digit recognising ANN to SNN from scratch using **LIF-Leaky model**.

MNIST is a large database of handwritten digits that is commonly used for training various image processing systems.

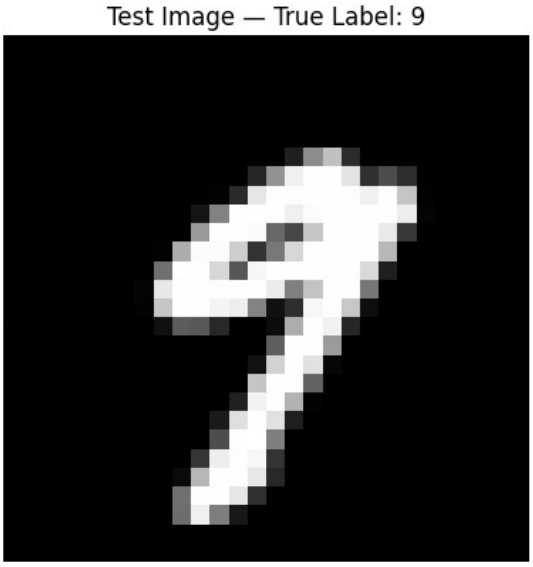


Figure 22: Image with true label 9 was tested which also looked like 7

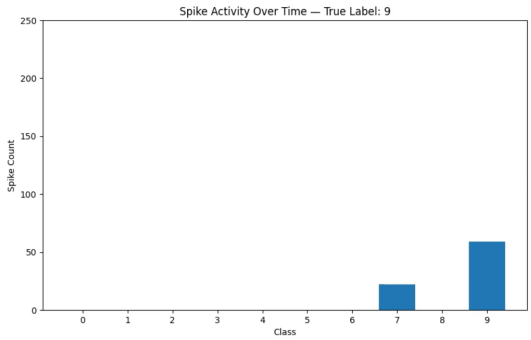


Figure 23: The Image gave spikes for 7 as well, max number of spikes were recieved for label 9

Poisson Encoding: We used Poisson Rate Coding. Here, brigher the pixel, more frequent is the spike. The plot given alongside shows this correlation.

Results: Our model was able to 96.84 % accuracy on 10,000 MNIST sample test on training for 10 epochs. Shallow ANNs need more epochs (20–50) to hit 97–98 % and deep CNNs requiring 10 - 20 epochs to reach accuracy of 99%.

Thus, SNNs show competitive accuracy with far fewer weight-updates (10 vs. 20–50 epochs) but on today's GPUs/CPUs they train much slower per epoch than traditional ANNs/CNNs causing longer training times.

7.3 Conversion Methodology

The ANN-to-SNN conversion approach was inspired by the paper *"Playing Atari using Spiking Neural Networks"*. We followed the weight transfer method outlined in the paper with the following steps:

1. **Train a DQN (ANN)**: A convolutional DQN was trained using standard reinforcement learning techniques on the Atari Breakout environment. The network used ReLU activations.
2. **Input Encoding (Poisson Rate Coding)**: Input frames were encoded into spike trains using Poisson encoding, where the probability of a spike at each time step is proportional to the pixel intensity.
3. **Weight Transfer and Scaling**:
    - Weights from the trained ANN were directly copied into an SNN architecture.
    - To compensate for reduced spiking activity in deeper layers, we applied layer-wise scaling:
        - Layer 1: weights scaled by 10×
        - Layer 2: weights scaled by 100×
4. **Spiking Neuron Model**: ReLU activations were replaced with Leaky Integrate-and-Fire (LIF) neurons that accumulate membrane potential over time and fire when a threshold is crossed.
5. **Action Selection**: The output layer spikes were counted over a simulation window, and the action corresponding to the neuron with the highest spike count was selected.

7.4 Results and Observations

- The ANN model achieved an average score of **169** over the last 50 episodes, with a peak score of **290**, outperforming the human baseline of 32.
- The converted SNN retained comparable performance when evaluated over multiple time steps.
- The conversion process avoided complex gradient computation, making it efficient and more robust for real-time simulation.

8 Conclusion

This project aimed to explore the viability of Spiking Neural Networks (SNNs) in reinforcement learning tasks, particularly in the Atari Breakout environment. The goal was to analyze various training strategies and determine a scalable, efficient way to deploy SNNs without compromising performance.

8.1 Training DQN without Convolution Layers but Higher Time-Steps

These were the final results for SNN converted model:

- The model training time was too much.
- It took almost 2.5 hrs to train for just to train 50 episodes.

Key Learnings:

- Direct SNN training using surrogate gradients or STDP is highly unstable in deep RL setups.
- Surrogate-based methods are sensitive to hyperparameters and computationally expensive due to multi-step spike simulations.

Final Results:

Overall, the project highlights the feasibility of using ANN-to-SNN conversion for RL tasks and positions it as a strong alternative to biologically-inspired but less scalable training methods.

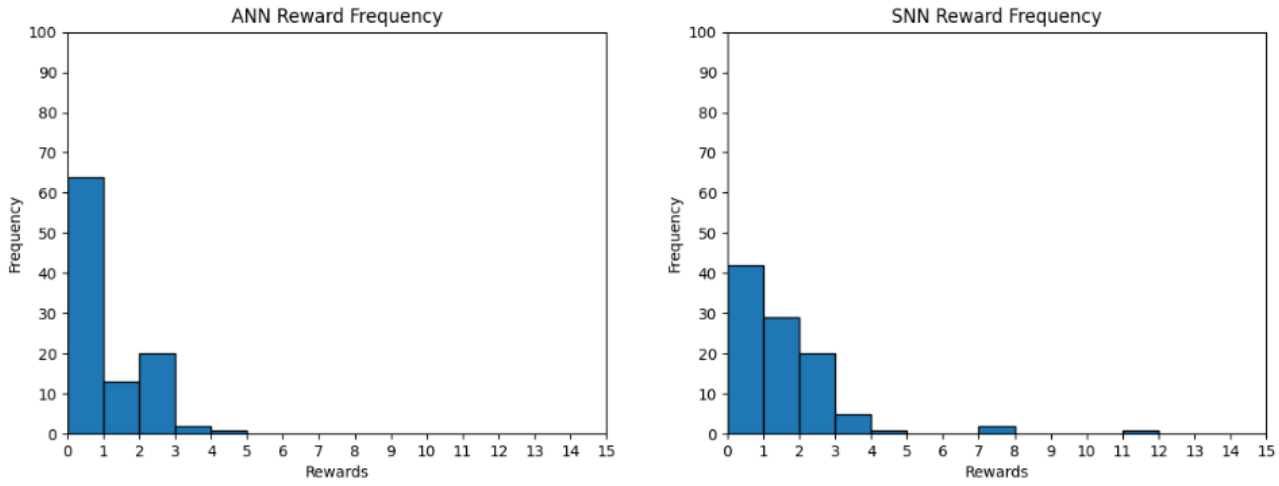


Figure 24: (Left) Reward distribution for BreakoutV4 with ANN (Mean = 0.63).
(Right) Reward distribution for SNN (Mean = 1.13).