Low-Level Learning: A C++ Engine for Foundational AI Techniques

Nick Brenner (nlb74), Alex Kozik (ajk33), Kevin Weng (kw444)

May 2025

AI Keywords: Reinforcement Learning, Deep Learning, Policy Gradient Methods, Multi-Armed Bandits, Autonomous Agents, Low-Level Optimization

Application Setting: Binary discriminative models and reinforcement learning; ML Optimization under constrained environment

Links:
Watch Project Video

1 Project Description

1.1 Motivation

Low-level optimization in Artificial Intelligence (AI) is of paramount importance in resource-constrained settings. In a world dominated by big data and deep learning, performance is often bottlenecked by resource constraints and a lack of access to compute power, as opposed to model complexity or data quality. For instance, in tactical edge computing system deployed by the military [11] or in our everyday mobile devices becoming equipped with machine learning [2], computational resources constrain the ability to use modern state-of-the-art AI models due to processing power, memory, and communication limitations. C++ has emerged as a predominant language for its static-typing and fine-grained memory control, and statistics such as the CPPJoules Energy Measurement Tool [8], based on Intel's Running Average Power Limit (RAPL) interface, have emerged predominant in recent literature to directly quantify real-world energy consumption (which is rapidly increasing for recent large-language models). We seek to implement a fine-grained C++ library, evaluated on all paradigms of AI, to demonstrate C++ as a tenable approach to accomplish programatically a diverse array of AI experiments.

1.2 Project Introduction

Here, we initially proposed a project that aimed to implement a low-level, optimized C++ library that is equipped to run complex AI algorithms, and we evaluate its robustness using experiments from all spheres of AI, including Machine, Reinforcement, and Deep Learning. **Section** 1 will be dedicated to introducing all of these models and explaining their underlying implementations. In **Section** 2, we discuss the non-trivial Artificial Intelligence applications we did, each of which combines and makes use of models in a non-trivial way, so as to evaluate and confirm the rigor of our implementation. A summary of our implemented models can be found in Table 1:

Category	**Algorithm**	**Description / Use Case**
Linear Models	Perceptron	Early binary classifier using a linear decision boundary.
	Linear Regression	Predicts continuous values with a linear relationship.
	Logistic Regression	Binary classification using the sigmoid function.
	Vanilla SVM	Maximizes margin for linear classification.
Non-Linear Extensions	Kernel SVM	Non-linear SVM using kernel trick (e.g., RBF).
	Random Fourier SVM	Approximates kernel SVM using random Fourier features for speed.
Unsupervised Learning	k-Means Clustering	Partition data into k groups based on similarity.
	PCA (Principal Component Analysis)	Dimensionality reduction and feature decorrelation.
Instance-Based Learning	k-Nearest Neighbors (k-NN)	Classifies based on majority vote of nearest neighbors.
Neural Approaches	Neural Networks	Deep learning models with multiple layers and nonlinearities.
Reinforcement Learning	ε-Greedy MAB	Balances exploration and exploitation via random action.
	UCB (Upper Confidence Bound) MAB	Selects arms using upper confidence intervals for efficient exploration.
	Deep Q-Network (DQN)	Uses deep neural nets to approximate Q-values for decision making.
	Policy Gradient (REINFORCE)	Optimizes policy directly using Monte Carlo estimates.

Table 1: Overview of implemented algorithms in the custom machine learning library.

We implement all of the above from scratch in c++ in hopes of better understanding what low-level optimization looks like in practice and to gain finer-grained control over memory and runtime optimizations.

1.3 Binary-Classification/Regression Algorithms

We first implemented standard binary-classification algorithms with linear decision boundaries, such as Perceptron, Logistic Regression, and Vanilla SVM. Vanilla SVM is the most sophisticated of these as it maximizes the hyperplane between the two classes. Because practical use cases are not entirely linearly separable, we use hinge loss with a tunable parameter $C \in \mathbb{R}$ to allow for "hard" versus "soft" decision boundaries to be learned. As an extension to this, we implemented kernelization for SVMs to allow for nonlinear data to also be separated. This leverages the kernel trick, where some function ϕ gives the inner product between two vectors $v, w \in \mathbb{R}^d$ for some $d \in \mathbb{N}$, rather than defining an explicit higher-dimensional projection. This allows for non-linearly-separable data to be projected into a higher dimension, where it may be separable by a linear hyperplane.
For regression, we implement linear regression, which fits a linear model to the data by optimizing against the Mean-Squared Error (MSE).

1.4 Unsupervised/Instance-Based Learning Algorithms

We implemented both k-NN and k-means. In particular, k-NN classifies a new test point by computing the binary label of the k nearest neighbors and taking a majority vote. Conversely, k-means does not need labeled data, and starting with randomly initialized centroids, it computes the means of the k nearest data points, shifts the centroid to that mean, and continues until the centroids converge (typically occurs within 500 iterations).
Principal Component Analysis is useful for projecting data down from a higher dimension. The eigenvectors of some linear transformation $A \in \mathbb{R}^{n \times n}$ give the principal components, which correspond to eigenvalue magnitude. For numerical stability purposes, we compute the top d eigenvectors using the more sophisticated Householder Reflection [5] algorithm, as opposed to more standard approaches like Gram-Schmidt. This is because normalization of vectors can become extremely sensitive to small numbers, and convergence becomes severely inhibited. Using the top d eigenvectors (principal components), a projection matrix $W \in \mathbb{R}^{d \times n}$ is formed, and $WA \in \mathbb{R}^{d \times n}$ induces a new linear transformation that projects data from the original dimension to a lower dimension $d < n$. The principal components are used due to preserving the highest amount of variance within the data.

1.5 Reinforcement Learning Algorithms

The goal of reinforcement learning is to learn some policy π_θ parameterized by θ where the agent acts over an environment with a predefined state and action space and learns how to act to optimize the accumulated reward.

1.5.1 Deep Q-Network

The Q function (or action-value function) is induced by a particular policy π. In particular, it is given as:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s, a_0 = a \right] \tag{1}$$

This allows us to precisely quantify an agent's performance as it gives the expected reward given that we are in a state s and take action a, and then we act in expectation thereafter. The optimal policy π^* has a Q-function is given by $Q^*(s, a) = \max_\pi Q^\pi(s, a) = Q^{\pi^*}(s, a)$, which we seek to learn.
In small environments (e.g. with discrete, small state and action spaces), the Q-function is actually tractable and can be computed directly. This would be called tabular Q-learning. Here, we implement the nonlinear variant, deep Q-learning [6], which attempts to learn the Q-function via a neural network. Given some state s_t, it is embedded as a d-dimensional vector and pushed through a neural network to obtain $Q(s, a), \forall a \in A$, where A is the action space.

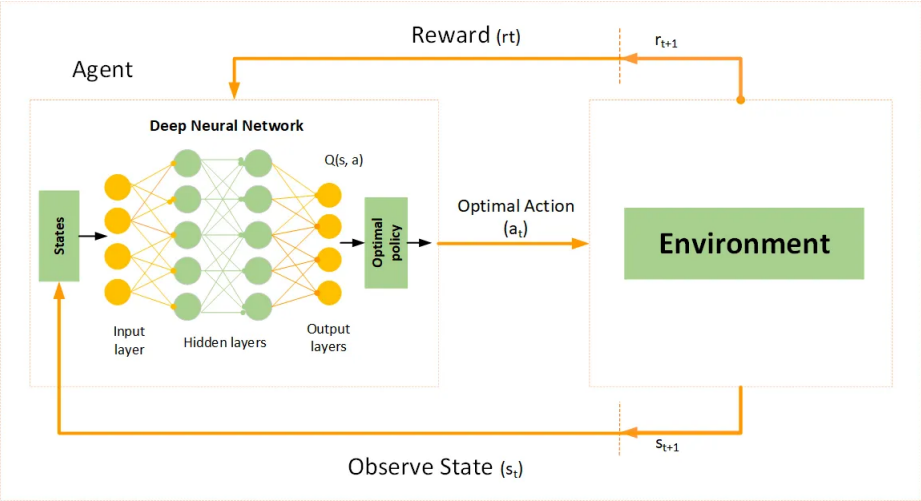


Figure 1: Deep Q-Learning graphic

We first initialize our neural network with randomly initialized weights θ. The network induces a policy π given by $\pi(s) = \text{argmax}_a \text{Q(s, a)}$. In our environment, we can unroll our policy to obtain $\{s_t, a_t, r_t\}_{t=1}^N$, where N is the length of the trajectory. Then, using 1, we compute the label $y_t = r_t + \gamma \max_a Q'(s_{t+1}, a'; \theta')$, which is an estimate of the optimal Q-function. Our optimization problem thus simplifies to the squared loss $(Q(s_t, a_t; \theta) - y_t)^2$, which we can do gradient descent over. There are two nuances here:

1. We do not use Q to compute y_t because then we have the 'moving target problem.' In particular, a network Q is trying to optimize against itself, which does not work in practice. To combat this, a target network Q' parameterized by θ' is used, and it is updated every fixed-length of iterations to match Q. [6]

2. When training, we use a replay buffer to sample past experiences $\{s_t, a_t, r_t\}$ uniformly. This is to prevent the network from forgetting how to act earlier on, especially later in training when those reward signals are no longer being seen.

1.5.2 Policy Gradient (REINFORCE)

With Deep Q-Networks, we induced a policy π with the learned intermediary $Q(s, a)$. Policy Gradient attempts to learn the gradient of the policy with estimates of the gradient using REINFORCE [10], and then to do gradient ascent directly to maximize the objective.
In particular, we rollout a trajectory τ to obtain $\tau = \{s_t, a_t, r_t\}^N$. Then, at each timestep t, we have the 'reward-to-go' $G_t = \sum_{k=t}^T \gamma^{k-t} r_k$. We define the objective as $J(\theta) = \mathbb{E}_{\pi_\theta}[G_t]$, i.e. maximizing in expectation the cumulative reward we can get. According to the Policy-Gradient Theorem [9], $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t \mid s_t) G_t]$.
We can thus train a neural network π_θ parameterized by θ so that given some state s_t, the outputs are the probabilities associated with taking each a_t. Using REINFORCE, we optimize θ by estimating $\nabla_\theta J(\theta)$ and doing gradient ascent.

1.5.3 ϵ-Greedy Multi-Armed Bandit

Multi-Armed Bandits (MABs) refer to a setting where there is an agent that has to pull an arm $\{a_1, \ldots, a_k\}$, each of which return variable rewards r_t. Unknowing of the reward distributions for each arm, the agent is tasked with learning a policy of which arms to pull in order to maximize reward.
ϵ-Greedy MABs are a simple, yet effective MAB that are parameterized with some $\epsilon \in [0, 1]$. At timestep t, the MAB pulls a random arm with probability ϵ and keeps track of the average rewards from each arm it sees. With probability $1 - \epsilon$, the agent acts greedily, pulling the arm that has given it the highest average reward so far. In summary, the agent is doing exploration, and aims to eventually learn the distribution of each reward arm effectively enough so as to commit to a singular arm. We implemented ϵ-Greedy MABs with a decay factor so that as the agent better learns the reward distributions, ϵ decays and the agent acts more greedily.

1.5.4 Upper Confidence Bound Multi-Armed Bandit

Upper Confidence Bound (UCB) MABs are a more sophisticated approach to learning a policy for which arms to pull. Just like ϵ-Greedy, UCB keeps track of the estimated reward for arm i up to timestep t, denoted as $\hat{\mu}_i(t)$. But, the agent also keeps track of the number of times it has pulled arm i, denoted as $N_i(t)$, and computes the upper

confidence bound $\mathrm{UCB_i(t)} = \hat{\mu}_\mathrm{i}(\mathrm{t}) + \sqrt{\frac{2 \ln \mathrm{t}}{\mathrm{N_i(t)}}}$. At timestep t, the agent selects a_t for which $a_t = \max_i \mathrm{UCB_i(t)}$.
Per the formulation above, it can be seen that the more times an arm is pulled, the higher $N_i(t)$ becomes, thereby reducing the upper-confidence bound. Over time, the upper-confidence bounds will become closer to the true means μ for each arm i.

1.6 Deep Learning Algorithms

For Deep Learning, we implement standard neural networks, in particular, the Multi-Layer Perceptron (MLP). We limit the scope of our architecture to be alternating linear and nonlinear layers, ending with a linear layer. We allow for variable lengths in input and layer lengths. For weight initializations, we use the popular He Initializations [4], which work well with sigmoid activation layers.
For optimization, we implemented back propagation using the algorithm [1] presented in Figure 2:

Algorithm Backward Pass through MLP (Detailed)

1: **Input:** $\{\mathbf{z}^{[1]}, \ldots, \mathbf{z}^{[L]}\}$, $\{\mathbf{a}^{[1]}, \ldots, \mathbf{a}^{[L]}\}$, loss gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[L]}}$
2: $\delta^{[L]} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[L]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[L]}} \frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{a}^{[L]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[L]}} \odot \sigma^{[L]'}(\mathbf{a}^{[L]})$ ▷ Error term
3: **for** $l = L$ **to** 1 **do**
4: $\quad \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{W}^{[l]}} = \delta^{[l]} (\mathbf{z}^{[l-1]})^T$ ▷ Gradient of weights
5: $\quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$ ▷ Gradient of biases
6: $\quad \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l-1]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l-1]}} = (\mathbf{W}^{[l]})^T \delta^{[l]}$
7: $\quad \delta^{[l-1]} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[l-1]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[l-1]}} \frac{\partial \mathbf{z}^{[l-1]}}{\partial \mathbf{a}^{[l-1]}} = ((\mathbf{W}^{[l]})^T \delta^{[l]}) \odot \sigma^{[l-1]'}(\mathbf{a}^{[l-1]})$
8: **end for**
9: **Output:** $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1:L]}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[1:L]}}$

Figure 2: Backpropagation algorithm through an MLP.

The loss gradient with respect to the final layer outputs is given to specify which loss to optimize. For optimization, we implemented mini-batch stochastic gradient descent by allowing the gradients to accumulate and normalizing over the batch size.

2 Evaluation

We designed and developed four distinct and nontrivial AI experiments to evaluate the capabilities of each of our models above and to verify the rigor of our low-level design.

2.1 N-Dimensional TicTacToe Agent

We developed an environment for both 3×3 and 4×4 TicTacToe and trained with both of our policy methods DQN and Policy Gradient to obtain autonomous agents to play the game. In particular, we developed a reward function that would reward signals of different magnitudes based on blocks, wins, losses, and draws. A summary of the models evaluated with this experiment is given in Table 2.

Model	How it is Used in the Experiment
DQN (Deep Q-Network)	The DQN is used to train an agent to play TicTacToe by approximating the Q-function.
Policy Gradient (REINFORCE)	The REINFORCE algorithm is used to optimize the policy directly by adjusting the probability distribution of actions based on the rewards received.
Neural Networks (Backpropagation)	Neural networks, using backpropagation, are employed to represent the Q-function for DQN and the policy $\pi(a_t \mid s_t)$ for Policy Gradient.

Table 2: Models Used in TicTacToe AI Experiment and Their Application

We examined the affects of ϵ_{decay}, $\mathcal{B}$ = Batch Size, α = Learning Rate, and $\mathcal{A}$ = Q-Network Architecture We now discuss each experiment. In the description for each experiment, we will start by stating the parameters that were fixed and the values of the parameter that was varied. We then will interpret the results of the experiment.

2.1.1 ϵ_{decay} affect on training trajectory

First, we examined the effect of ϵ_{decay} on model performance over 1000 training episodes. The results can be seen in Figure 3:

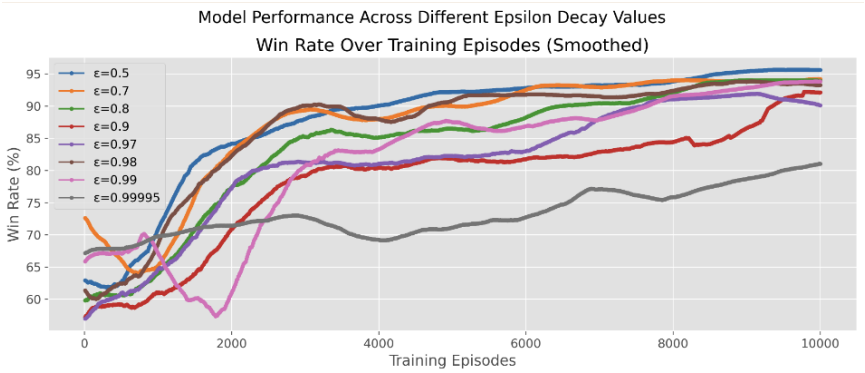


Figure 3: Enter Caption

In this experiment, the following parameters were fixed: $\mathcal{B} = 64$, $\alpha = 0.0001$, $\mathcal{U} = 10$, $\epsilon_{\text{start}} = 0.9$, $\epsilon_{\text{end}} = 0.001$, $\gamma = 0.9$, $\mathcal{E} = 10000$, $\mathcal{A} = 18 \to 32 \to \text{ReLU} \to 32 \to \text{ReLU} \to 9$. ϵ_{decay} varies in:

$$\epsilon_{\text{decay}} \in \{0.5, 0.7, 0.8, 0.9, 0.97, 0.98, 0.99, 0.99995\}$$

The agents with smaller ϵ_{decay} values—specifically 0.5, 0.7, and 0.8—demonstrate the fastest improvement in win rate. These agents quickly transition from exploration to exploitation, allowing them to capitalize on learned strategies early in training. As a result, their win rates rise sharply and plateau at high values (above 90%) within the first few thousand episodes. This rapid improvement with minimal exploration aligns with Tic-tac-toe's inherent simplicity. Given the game's small state space and straightforward winning patterns, extensive exploration is unnecessary. The agents with intermediate decay values, such as 0.9, 0.97, 0.98, and 0.99, also achieve high win rates, but their improvement is more gradual. These agents balance exploration and exploitation more conservatively, leading to a slower but steady increase in performance. They eventually reach similar win rates as the fastest-rising agents, but require more episodes to do so. On the other hand, the agent with the slowest decay ($\epsilon_{\text{decay}} = 0.99995$) exhibits a very slow rise in win rate and fails to reach the performance levels of the other agents within the allotted training episodes. This highlights the drawback of excessive exploration in Reinforcement Learning Models: the agent spends too much time trying random actions and not enough time refining and exploiting effective strategies.

2.1.2 $\mathcal{B}$ affect on training trajectory

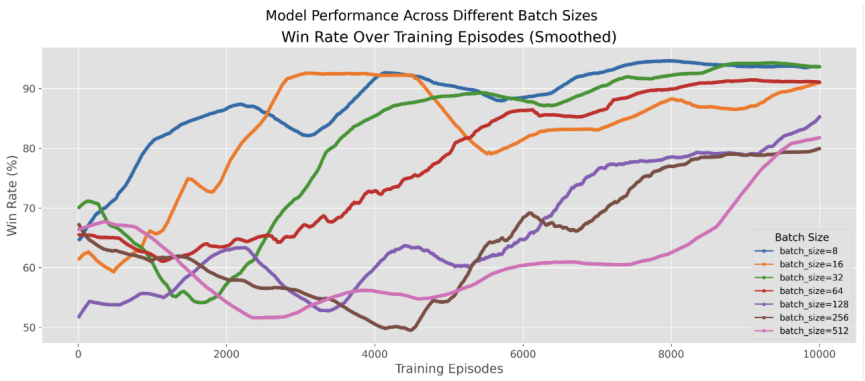


Figure 4: Enter Caption

In this experiment, the following parameters were fixed: $\epsilon_{\text{decay}} = 0.999$, $\epsilon_{\text{start}} = 0.9$, $\epsilon_{\text{end}} = 0.001$, $\mathcal{U} = 10$, $\alpha = 0.0001$, $\gamma = 0.9$, $\mathcal{E} = 10000$, $\mathcal{A} = 18 \to 32 \to \text{ReLU} \to 32 \to \text{ReLU} \to 9$. The batch size $\mathcal{B}$ was varied as

$$\mathcal{B} \in \{8, 16, 32, 64, 128, 256, 512\}$$

The results show that batch size has a significant effect on the agent's learning trajectory and final performance: Smaller batch sizes ($\mathcal{B} = 8, 16, 32$) lead to the fastest initial rise in win rate, with $\mathcal{B} = 8$ and $\mathcal{B} = 16$ reaching high win rates (above 90%) within the first 4000–6000 episodes. These agents are able to quickly adapt their policies, likely due to more frequent updates and higher variance in gradient estimates, which can help escape local optima early in training. Moderate batch sizes ($\mathcal{B} = 64, 128$) also achieve high win rates, but their improvement is more gradual. They eventually reach similar performance to the smallest batch sizes, but require more training episodes to do so.
Larger batch sizes ($\mathcal{B} = 256, 512$) result in much slower learning. These agents exhibit a delayed increase in win rate and, within the allotted training episodes, do not reach the same level of performance as agents trained with smaller batches. This is likely because larger batches provide more stable but less frequent updates, which can slow down the learning process and reduce the agent's ability to quickly adapt to new information.

2.1.3 α affect on training trajectory

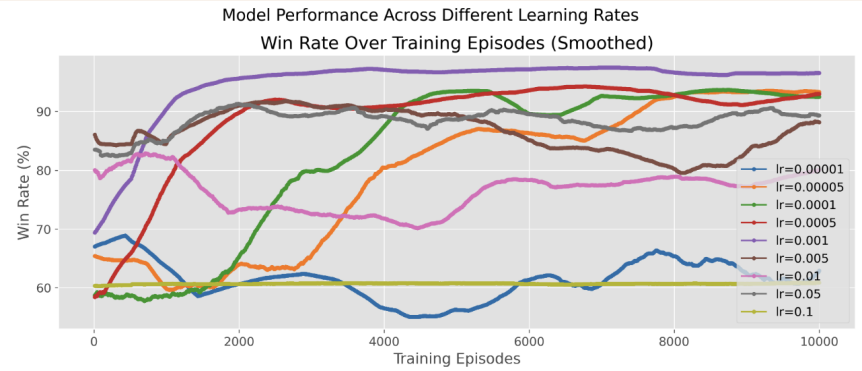


Figure 5: Enter Caption

In this experiment, the following parameters were fixed: $\mathcal{B} = 64$, $\epsilon_{\text{decay}} = 0.9$, $\epsilon_{\text{start}} = 0.9$, $\epsilon_{\text{end}} = 0.001$, $\mathcal{U} = 10$, $\gamma = 0.9$, $\mathcal{E} = 10000$, $\mathcal{A} = 18 \to 32 \to \text{ReLU} \to 32 \to \text{ReLU} \to 9$. The learning rate α was varied as

$$\alpha \in \{0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1\}$$

The choice of learning rate had a substantial impact on both the speed of training and the final performance.
Moderate learning rates ($\alpha = 0.0005, 0.001, 0.005$) achieve the best performance, with win rates rising quickly and plateauing above 90%. These values allow the agent to make sufficiently large updates to learn efficiently, without causing instability. Very small learning rates ($\alpha = 0.00001, 0.00005, 0.0001$) result in much slower learning. The win rates for these agents increase only gradually, and do not reach the same performance. This is likely because the updates are too small for the agent to effectively adapt its policy.
Very large learning rates ($\alpha = 0.01, 0.05, 0.1$) lead to poor or unstable performance. The win rates for these agents either stagnate at a low value or fluctuate significantly, indicating that the updates are too aggressive and may cause the agent to overshoot optimal solutions or fail to converge.

2.1.4 $\mathcal{A}$ affect on training trajectory

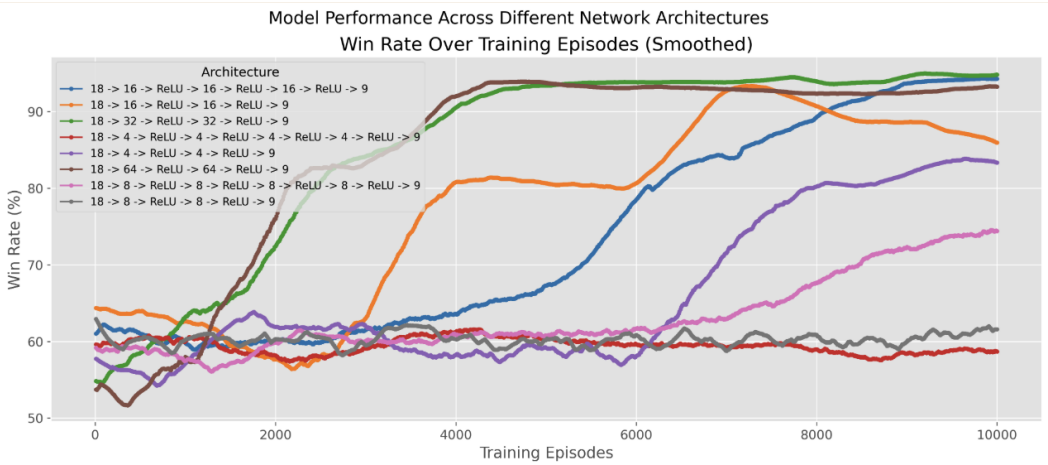


Figure 6: Enter Caption

In this experiment, the following parameters were fixed: $\mathcal{B} = 64$, $\epsilon_{\text{decay}} = 0.99$, $\epsilon_{\text{start}} = 0.9$, $\epsilon_{\text{end}} = 0.001$, $\mathcal{U} = 10$, $\alpha = 0.0001$, $\gamma = 0.9$, $\mathcal{E} = 10000$. The network architecture $\mathcal{A}$ varies as portrayed in the legend.
The network architecture had a substantial impact on both the speed of training and the final performance. Larger and moderately sized architectures (such as $18 \to 64 \to \text{ReLU} \to 64 \to \text{ReLU} \to 9$ and $18 \to 32 \to \text{ReLU} \to 32 \to \text{ReLU} \to 9$) achieve the highest win rates, rising quickly and plateauing above 90%. These architectures have enough capacity to represent effective policies and learn efficiently from experience.
Very deep or very narrow architectures (such as those with multiple layers of size 4 or 8) tend to learn more slowly and plateau at lower win rates. This suggests that either too little capacity (narrow networks) or excessive depth (without sufficient width) can hinder learning in this environment.
Intermediate architectures (e.g., $18 \to 16 \to \text{ReLU} \to 16 \to \text{ReLU} \to 9$) show moderate performance, with win rates rising steadily but not reaching the highest levels achieved by the wider networks.

Overall, these results illustrate that: Wider and moderately deep networks are best suited for this task, enabling fast and effective learning. Very deep or very narrow networks may struggle to learn optimal policies, either due to insufficient capacity or optimization difficulties.
This highlights the importance of tuning the network architecture in deep reinforcement learning. An appropriately chosen architecture allows the agent to efficiently learn from experience and achieve high performance, while architectures that are too small or too deep may impede learning and result in suboptimal performance.

2.2 REINFORCE vs. DQN

Our REINFORCE implementation featured a similar architecture but with larger hidden layers (64 neurons each) and a lower learning rate of 0.0001 to improve training stability for this policy gradient approach. As is standard with REINFORCE, it does not use experience replay, discount factors, or epsilon-greedy exploration mechanisms. Results are given in Figure 7:

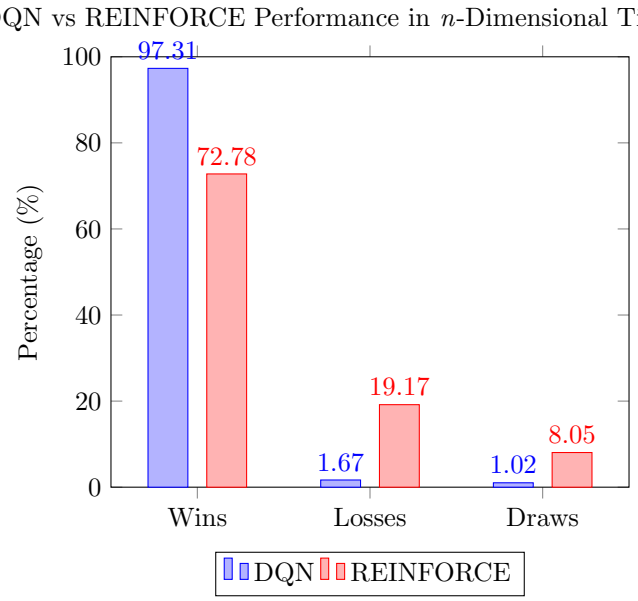


Figure 7: Performance comparison between DQN and REINFORCE agents in n-dimensional Tic-Tac-Toe. DQN outperforms REINFORCE with a substantially higher win rate and lower loss/draw rates.

The experimental results show a clear performance advantage for DQN in the Tic-Tac-Toe environment. DQN achieved a 97.31% win rate with only 1.67% losses and 1.02% draws. In contrast, REINFORCE managed a 72.78% win rate with significantly higher loss (19.17%) and draw (8.05%) rates. This represents a substantial performance gap of nearly 25 percentage points in win rate.

The superior performance of DQN can be attributed to several factors inherent to the algorithm. First, DQN's value-based approach provides more stable learning signals in a deterministic game like Tic-Tac-Toe. Second, the experience replay mechanism enables more efficient use of training data by breaking correlations between consecutive samples. Finally, the structured ϵ-greedy exploration strategy more effectively balances exploration and exploitation compared to the policy stochasticity that REINFORCE relies on.

These results suggest that for discrete, deterministic environments with clear reward structures like Tic-Tac-Toe, DQN offers significant advantages over policy gradient methods like REINFORCE. Despite REINFORCE using a higher-capacity network, it was unable to match DQN's ability to efficiently learn the optimal policy for this particular task.

2.3 Music Recommender

In this experiment, we use the Kaggle Spotify Dataset [7] in order to create an agent that responds to user feedback and adapts music recommendations accordingly. First, we use k-means to cluster the dataset based on 7 of the numerical features, described in Table 3:

Feature	**Description**
Valence	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track (higher means more positive).
Acousticness	Confidence measure (0.0 to 1.0) of whether the track is acoustic.
Danceability	Describes how suitable a track is for dancing based on tempo, rhythm stability, beat strength, and regularity.
Energy	Represents a perceptual measure of intensity and activity; ranges from 0.0 (low energy) to 1.0 (high energy).
Instrumentalness	Predicts whether a track contains no vocals; the closer to 1.0, the more likely it is instrumental.
Liveliness	Detects the presence of an audience in the recording; higher values indicate a more "live" feel.
Speechiness	Detects the presence of spoken words; values closer to 1.0 suggest more speech-like tracks.

Table 3: Descriptions of selected Spotify audio features from the dataset.

To obtain the clustering, we allow for different options in order to explore performance. The standard approach is to cluster directly on the 7-dimensional data points, and to then use the MAB (ϵ-greedy versus UCB) to learn a policy for how to select which cluster to recommend songs from. However, we also incorporate PCA in order to project the data down into a lower dimensional subspace, after which clustering may occur. In doing so, we are able to find the optimal dimension for the clustering that in turn impacts MAB performance. We also use PCA to visualize the data in 2-dimensions. A summary of the models evaluated with this experiment is given in Table 4:

Model	How it is Used in the Experiment
k-means	k-means is used to cluster music tracks based on extracted features. The algorithm groups the tracks into distinct clusters, which allows the recommender system to identify similar tracks. Each cluster represents a group of musically similar tracks.
PCA (Principal Component Analysis)	PCA is used to reduce the dimensionality of the feature space before applying k-means. It projects the data from 7 dimensions to 2 dimensions for easier visualization of the clustering process.
MABs (Multi-Armed Bandits)	Both ϵ-greedy and UCB (Upper Confidence Bound) strategies are used to learn which cluster (or "arm") to recommend to users. The bandit algorithms explore and exploit the different clusters based on user feedback.

Table 4: Models Used in the Music Recommender Experiment and Their Application

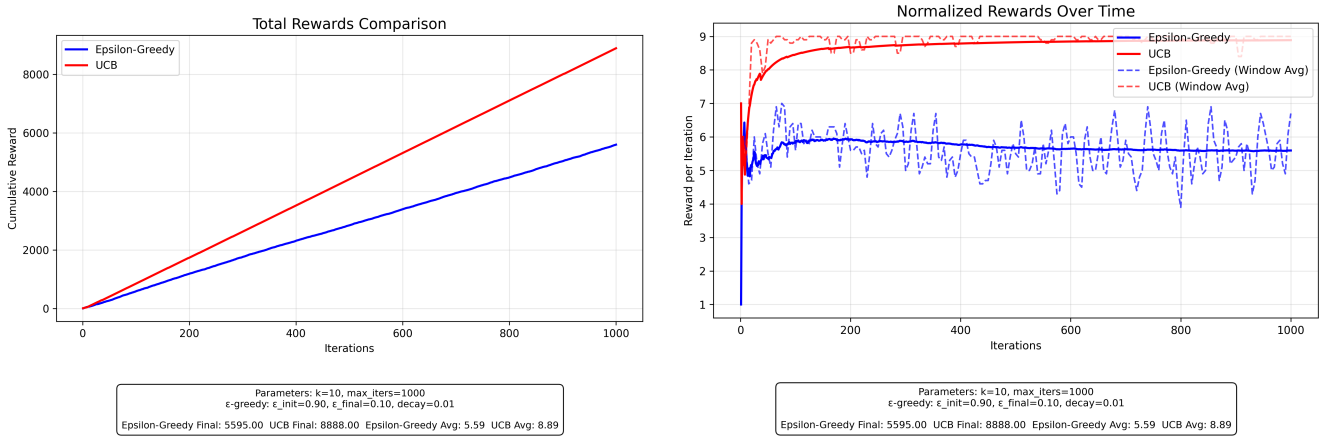
After clustering, the MAB pulls one of the arms (the clusters) to suggest a song for the user. The user responds with a reward from 1-10, which is used to update the MAB. The process continues, and over time, the MAB gradually learns the user's preferences and dynamically adapts its recommendations. In practice, such a recommender would be integrated with a user's device and track statistics over thousands of episodes. We devised a method to circumvent the need for human feedback and to approximate the quality of recommendations. First, we create a user vector $v \in \mathbb{R}^7$ where each $v_i \in [0, 1]$, indicating how much the user likes the i-th feature in 3. k-means clustering is performed, and then with each recommended song (denote as $w \in \mathbb{R}^7$, we compute the reward as follows:

$$r = 1.0 + 9.0 \left[\sigma \left(20.0 \cdot \left(\frac{v \cdot w}{||v||_2 ||w||_2} \right) - 0.55 \right) \right]$$

where σ denotes the sigmoid function and $\frac{v \cdot w}{||v||_2 ||w||_2}$ is the cosine similarity between the user vector v and song vector w. Due to the compactness of the data, we introduced the bias shift 0.55 to control the threshold at which the sigmoid function outputs values greater than 0.5. We scale by 20.0 to introduce a greater degree of variance and heightened sensitivity. Lastly, we multiply by 9.0 and add 1.0 in order to scale the reward signal between 1 and 10, as proposed in the setup above. With these reward signals, we run 1000 iterations and allow the MAB to explore the reward signals. We track two metrics:

1. Total reward (The total reward accrued since timestep 1
2. Average reward per timestep (The total reward, normalized over time t)

A summary of results can be seen in Figure 8:



(a) Total reward over time

(b) Normalized reward over time

Figure 8: Comparison of UCB and ϵ-Greedy MABs for reward exploration

Within 200 episodes, both ϵ-Greedy and UCB Have converged, as evidenced by their average reward per timestep plateauing. The figures also includes a sliding window over 10 episodes that calculates the current reward signal average over the last 10 timesteps, which is denoted by the dotted line on the right. As expected, UCB outperformed ϵ-Greedy as ϵ-Greedy committed to a suboptimal arm. By suboptimal, we refer to the fact that each arm has different mean reward signals, and some are noisier than others, so ϵ-Greedy did not explore the reward space sufficiently to commit to the optimal arm. This is because ϵ-Greedy is choosing from one of the $k = 10$ arms uniformly at random with probability ϵ, which is decaying over time. Conversely, the more sophisticated UCB algorithm also takes into account the **number** of times it has pulled an arm, which increases as uncertainty decreases over time. So, UCB is able to learn quicker which arm has the potential of giving the highest reward. We see this emirically due to UCB's fast convergence, as evidenced on the right.
These results above empirically confirmed that UCB can outperform ϵ-Greedy, particularly on our given task. We then proposed the research question of how k (the number of arms/clusters) impacts reward performance, and similarly how the projection dimension impacts performance. In the above experiment, we used $k = 10$ and $\text{dim} = 2$. Now, we vary k and the dimension for both ϵ-Greedy and UCB and seek to measure total reward, average reward, and **convergence time**. To measure convergence time, we use a different approach for ϵ-Greedy and UCB. This is because when $\epsilon > 0.5$, we are exploring with a greater probability than acting greedily, so convergence does not make sense in this case. A visual of the algorithm can be seen in Figure 9:

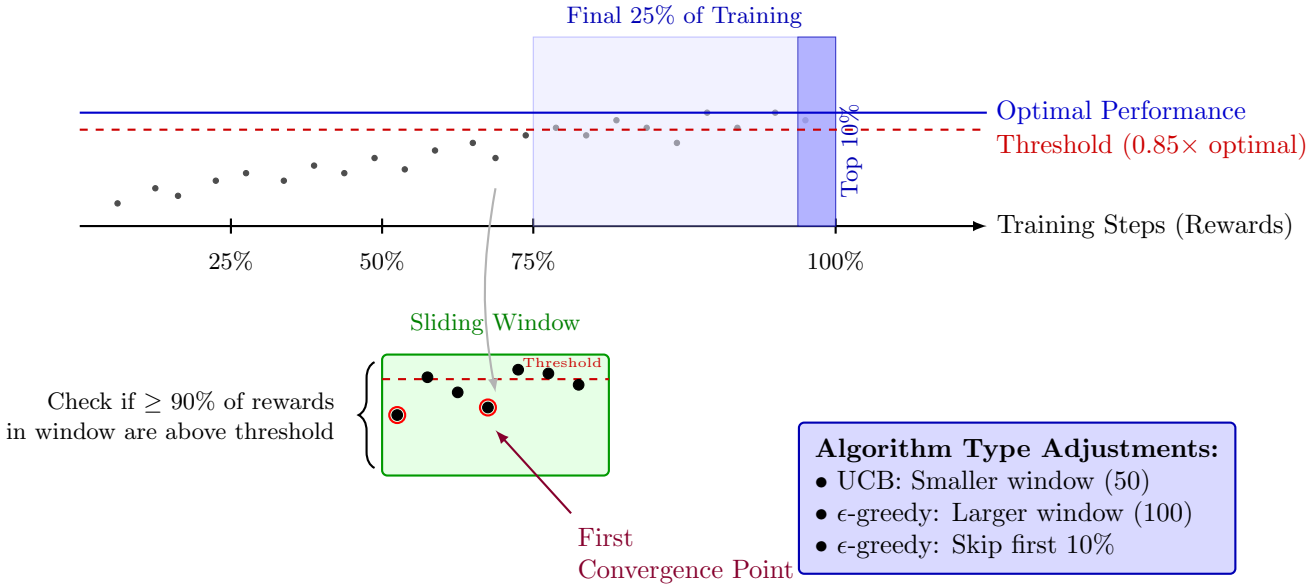
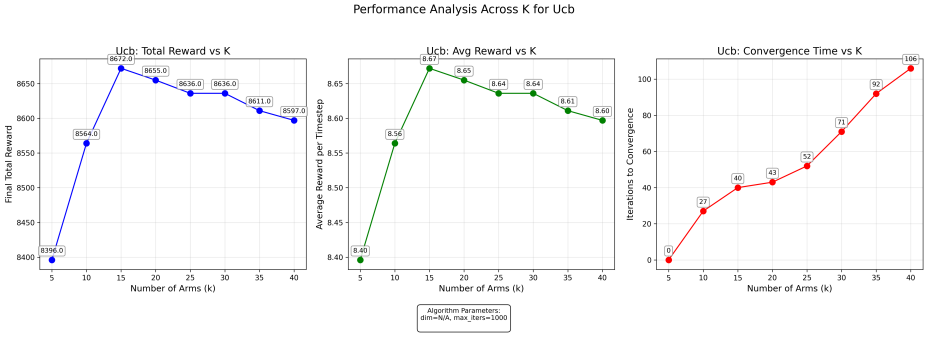


Figure 9: Sliding window convergence criteria used to determine training convergence.

Because convergence is final, we first examine the final 25% of iterations. From these, the top 10% of rewards are extracted to get a sense of the model's best performance. We then apply a threshold value of 0.85 to these

values. As seen in Figure 9, the optimal reward signals are marked as 'Optimal Performance,' and the dotted red line below denotes the 'Threshold.' We threshold due to the inherent noise in reward signals. Using a sliding window of 50 for UCB or 100 for ϵ-Greedy, we check if $\geq 90\%$ of the reward signals within the window are above the threshold, and if they are, then convergence is detected. Moreover, for ϵ-Greedy, we also skip the first 10% of reward signals due to the aforementioned exploration occurring early on in training. A summary of our results for the number of arms/clusters k experiment can be seen in Figure 10:

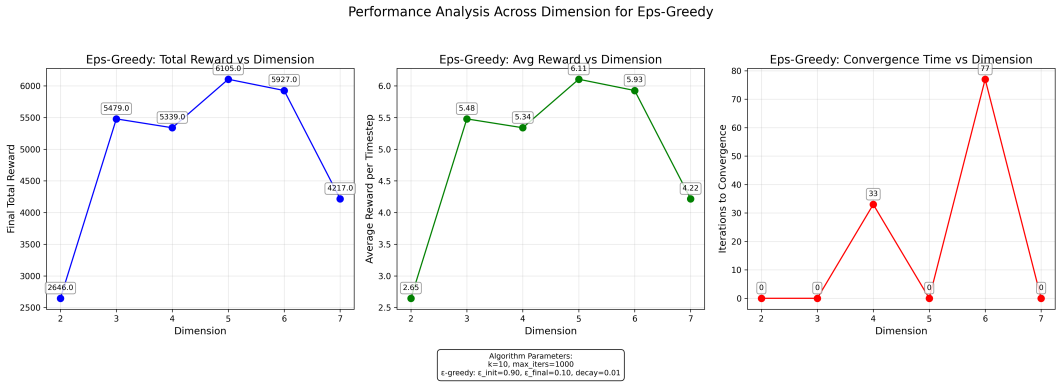
(a) ϵ-Greedy performance vs. k, 0 denotes no convergence for the Convergence Time



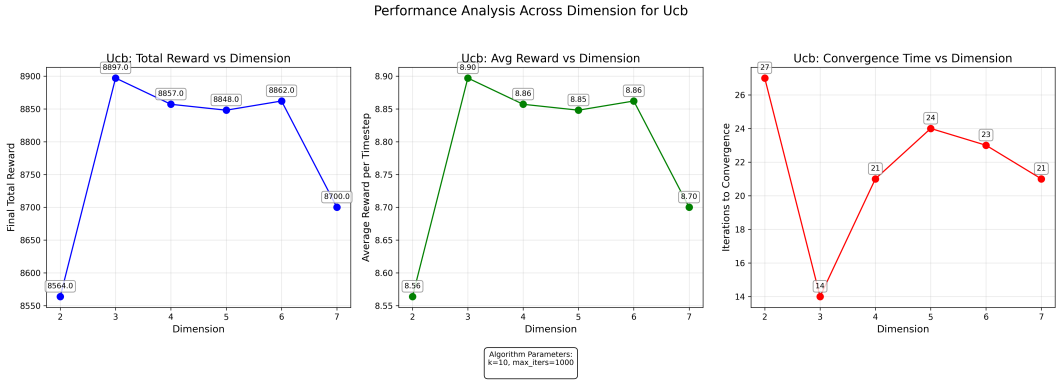
(b) UCB performance vs. k

Figure 10: ϵ-Greedy vs. UCB MAB performance vs. k

For ϵ-Greedy, we observe that a non-monotonic reward signal. This is because as we increase k, our ϵ-greedy exploration policy breaks down. The more arms to choose from, the more exploration is required and the less effective the uniform distribution exploration policy will be at learning. However, a higher k can sometimes result in further separation between the mean reward signal of each arm, which in turn makes ϵ-greedy more effective, which is why we do not see a consistent trend. For convergence time, we actually only see convergence for $k = 5, 20$, and 35, which further confirms the non-consitency and randomness in ϵ-greedy's performance, which makes it an undesirable choice. For the more sophisticated UCB, we have a "corner" in performance at $k = 15$, and clearly see a downwards trend afterwards, which indicates that $k = 15$ is the clear optimal choice. This choice of k likely separates out the reward signal means just enough, while also making exploration an easy enough task. For UCB, we always converge, and as k increases, convergence takes longer as expected, due to more means to explore.
Next, the results of varying the projection dimension are given in Figure 11:



(a) ϵ-Greedy performance vs. projection dimension, 0 denotes no convergence for the Convergence Time



(b) UCB performance vs. projection dimension

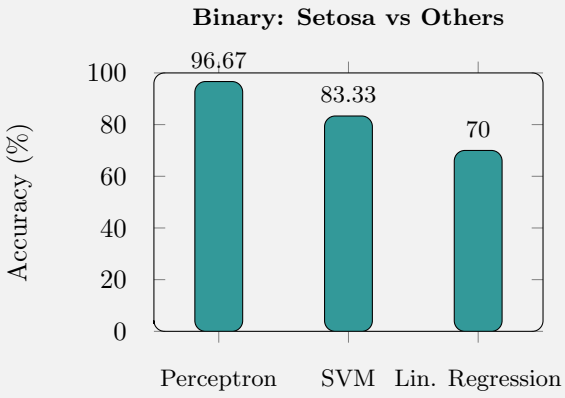
Figure 11: ϵ-Greedy vs. UCB MAB performance vs. projection dimension

In general, we see less strong trends as the dimension is varied because performance is contingent on PCA's accurate computation of the eigenvectors. For ϵ-Greedy, we see a peak in performance at a projection dimension of 5, which likely could have performed even better since it had not converged yet. For UCB, we see a clear boost in performance at a projection dimension of 3, which holds steady until the full dimension 7. This is likely due to the fact that the $2D$ projection space is likely too small of a subspace for the data to be separated, leading to poor clustering. Similarly, the full dimension of 7 likely starts to suffer from the curse of dimensionality, where there is sparsity of data and distance metrics like those used in k-means begin to break down, also leading to poor convergence. Overall, we are able to conclude that our music recommendation task is best accomplished using UCB (for faster/better convergence times) with $k = 15$ and a projection dimension $d \in \{3, 4, 5, 6\}$. Our results confirm empirically Reinforcement Learning Theory such as convergence times and better reward signals, and they illustrate phenomenons such as the curse of dimensionality in our particular use case.

2.4 Binary Classification

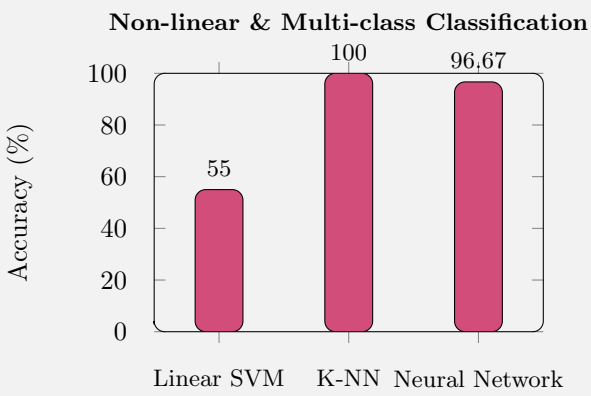
To evluate our binary classification/regression models, we use the popular Iris dataset [3]. This involves the classification of three different species of flowers, Setosa, Versicolor, and Verginica. This dataset is particularly useful because Setosa vs. Versicolor/Verginica is actually learnable via a linear decision boundary, which is useful for evaluating Perceptron and SVM. Likewise, the boundary between Versicolor/Verginica is nonlinear and requires nonlinear approaches, like k-NN. For reference, we report the SVM linear boundary on the nonlinear task as well, which underperformed at 55% accuracy. A summary of the models evaluated with this experiment is given in Table 12:

Binary Classification: Setosa vs Others



Model Details:
- **Perceptron:** 100000 iterations, 96.67% accuracy
- **SVM:** C=10.0, 100 iterations, lr=0.0001, 83.33% accuracy
- **Linear Regression:** On petal length, threshold $<$ 2.0 for Setosa, lr=0.0001, 70% accuracy

Multi-class Classification



Model Details:
- **Linear SVM:** Versicolor vs Virginica using petal dimensions, C=10.0, 100 max iterations, lr=0.0001, 55% accuracy
- **K-NN:** k=5, 100% accuracy
- **Neural Network:** 3-class, all 4 features, 2 hidden layers (64 units each), lr=0.01, 1000 epochs, batch size 64, 96.67% accuracy

Figure 12: Performance comparison of different machine learning models on the Iris dataset classification tasks. The left panel shows binary classification results for separating Setosa from other species. The right panel shows performance on the more difficult Versicolor vs. Virginica binary task and the full three-class classification problem.

As evidenced by the data, Perceptron and SVM were both able to effectively separate the linear decision boundary. We were able to use our Linear Regression model on the binary task as well by predicting the continuous petal length feature and applying a threshold $<$ 2.0 to convert to a positive label, negative otherwise. However, linear regression performs worse at 70% accuracy, which is likely due to the conversion to binary from continuous and the indirect approach.
For the non-linear cases, we can see that k-NN with just $k = 5$ was sufficient to classify between Versicolor and Verginica using both petal length and width, which is a nonlinear boundary. Similarly, neural networks with a simple architecture and mini-batch SGD optimization were able to classify between all 3 classes effectively using all 4 features.

2.5 Compiler Optimizations

One reason we chose to build our machine learning library in C++ is to take advantage of low-level compiler optimization options that are not available in Python, the standard language used for machine learning. Although PyTorch has bindings for C++, there are still parts of the code, such as loops, that will be exposed in Python and cannot be optimized. In our library, every part of the code can be optimized to generate highly efficient machine code tailored to the target hardware. These optimizations are especially impactful for compute-intensive operations such as matrix multiplcation and neural netwrk forward/backward passes.
To evaluate the effect of compiler optimizations on our project, we trained a DQN Agent to play Tic Tac Toe using the following configuration: $\mathcal{A} = 18 \to 32 \to \text{ReLU} \to 32 \to \text{ReLU} \to 9$, $\alpha = 0.0001$, $\epsilon_{\text{start}} = 0.9$, $\epsilon_{\text{end}} = 0.001$, $\mathcal{B} = 64$,$\gamma = 0.9$, $\mathcal{E} = 100$. We tested the following optimization levels:

- **No optimization flags** — Baseline, no compiler optimizations
- `-O1` — Basic optimizations
- `-O2` — More aggressive optimizations
- `-O3` — Enables function inlining, loop unrolling, vectorization
- `-O3 -ffast-math` — Adds aggressive floating-point math optimizations
- **All flags** — `-O3 -ffast-math -funroll-loops -flto -march=native -mtune=native -fomit-frame-pointer -finline-functions -fstrict-aliasing`. This combination applies maximum optimization:

- `-funroll-loops` — Unrolls loops to reduce branching and increase instruction-level parallelism
- `-flto` — Link Time Optimization enables optimizations across translation units
- `-march=native -mtune=native` — Generates code optimized for the local machine's architecture and tuning parameters
- `-fomit-frame-pointer` — Omits the frame pointer to free up a register
- `-finline-functions` — Aggressively inlines functions to reduce call overhead
- `-fstrict-aliasing` — Assumes that pointers to different types do not alias, enabling more aggressive reordering

As shown in the table, compiler optimizations have a significant impact on training time. With no optimizatino flags, training is about 26 times slower than when using the most aggressive optimization flags. These results show the importance of leveaging compiler-level optimizations when building performance-critical AI software.

Optimization Level	**Avg. Training Time (ms)**
No optimization flags	8345.10
`-O1`	713.20
`-O2`	392.30
`-O3`	375.90
`-O3 -ffast-math`	367.70
All flags	316.60

Table 5: Average training time (in milliseconds) over 10 runs for 100 episodes of training under different compiler optimization levels

In summary, were able to effectively implement a low-level C++ AI library that is robust to a multitude of applications, as seen in experiments 1-3. Likewise, due to C++'s exposure of memory control and management, we are able to have a finer-grained control over our library than an analogous implementation in an interpreted language like Python. As seen in experiment 4, this effect is acutely pronounced with compiler optimizations significantly impacting performance. In all, we hope to demonstrate that AI applications in C++ are robust and more importantly, economical in the current, power-hungry AI landscape.

References

[1] Cs 4782: Machine learning theory — week 1–2 slides, 2025. Accessed: 2025-05-13.

[2] Hanan Elayan, Mohamed Elhoseny, N. Kumar, and Seungmin Rho. Edgemlbalancer: A self-adaptive approach for dynamic model switching on resource-constrained edge devices. *arXiv preprint arXiv:2502.06493*, 2024.

[3] Ronald A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[5] Alston S. Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342, 1958.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[7] Maharshi Pandya. Spotify tracks dataset. `https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset`, 2022. Accessed: 2025-05-12.

[8] Elizaveta Solovyeva, Albert Sokolov, Nikita Sidorov, Yury Yanovich, Irina Makarova, and Alexander Panov. Cppjoules: A tool for measuring energy consumption of c++ programs. In *Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 123–134. IEEE, 2024.

[9] R. S. Sutton, R. T. McAllister, and D. Precup. Policy gradient methods for reinforcement learning with function approximation. *NIPS*, 12:1057–1063, 2000.

[10] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[11] Xinyu Zhao, Zheng Li, Lichen Song, Qingyi Liu, Chuan Zhang, Wenjie Wang, Xiangyu Zhang, Jinyuan Wu, Depeng Jin, and Jie Tang. On accelerating edge ai: Optimizing resource-constrained deployments. *arXiv preprint arXiv:2501.15014*, 2024.