**DataPredict: A General Machine, Deep, Reinforcement Learning Library For Lua-Native Games**

**Aqwam Harish Aiman**

**Abstract**

Machine learning, deep learning and reinforcement learning frameworks such as scikit-learn, TensorFlow, PyTorch and Stable Baselines allow researchers to train models using real-world datasets. However, there is a noticeable gap for game-related frameworks. This has led to difficulties in producing game-related machine learning, deep learning and reinforcement learning research as researchers are forced to integrate frameworks whose programming languages are incompatible with the game engines. Aiming at mimicking scikit-learn design, DataPredict offers a way of implementing machine learning, deep learning and reinforcement learning into Lua-native game engines like Roblox Studio.

**Introduction**

Machine learning, deep learning and reinforcement learning usage have become increasingly common in general real-world applications. Many frameworks, like scikit-learn and PyTorch, speed up the implementation of these models for general real-world applications. However, this focus has led to neglect in simulated worlds that mimic real-world interactions, such as games.

The gaming industry is estimated to be worth over $184 billion. This presents a significant opportunity for commercialising such models on a large scale. In addition, gaming platforms like Roblox that support user-generated content provide a significant amount of data that could be used to train these models. Although model training on this user-generated content has been conducted, very few have leveraged game-generated data to build in-game models that adapt to players in real-time. If done correctly, the in-game models could allow games to retain players much longer, earn higher revenue and have higher player-returning-power.

As such, this paper introduces DataPredict, a Lua library that is specifically designed for games written in Lua. Although DataPredict can be used for non-game-related contexts, its aim is to exploit game environments for model training.

**Background**

Most of the machine learning, deep learning and reinforcement learning frameworks are commonly written in Python. Because of Python’s ease of use, many practitioners often opt to build their code on top of these existing frameworks. This has led to an explosion of repositories that are reliant on these frameworks, creating a significant portion of the machine learning, deep learning and reinforcement learning communities.

Meanwhile, most of the biggest game engines do not rely on Python. For game engines that could simulate 3D physics, they are generally written in performant but less popular programming languages such as C++, C# and C in Unity Editor, Unreal and Godot Engine. Surprisingly, a game engine called Roblox Studio opted for Lua, which is less performant due to its single-threaded nature. As such, Roblox had to make major modifications to the original Lua programming language so that Roblox Studio could perform multi-threaded processes.

The incompatibility between the machine learning world and the game engine world has led to sparse research combining both machine learning and games. This is because workarounds are needed for cross-language communication, which requires additional resources and effort to set up. This example can be seen from Unity’s MLAgents, where it attempts to bridge the gap between C++ and Python since Unity is based on C++ and PyTorch is based on PyTorch. This creates a drawback that the resulting code becomes too inflexible to use as the practitioners must handle three aspects of codebases: the surface-level API of the game engine, the cross-platform communication and the underlying neural network structure running on PyTorch.

Currently, cross-language communication may introduce latency, restrict the ability to perform in-game training and cause code maintenance issues. Consequently, this may lead to flawed experimentations, training and research conclusions that could affect the overall research.

This incompatibility also led to an added effect on commercial adoption of using machine learning, deep learning and reinforcement learning in games. This is because more resources are needed to handle multiple programming languages, leading to commercial entities opting out of taking advantage of these models in games. This leads to underutilising machine learning, deep learning and reinforcement learning in games since a lot of data is generated from games alone but is not being used effectively.

As a result, DataPredict was created to tackle the cross-language communication issues and make it native to the game engines.

**Model Selection Criteria**

Currently, DataPredict offers over 50 machine learning, deep learning and reinforcement learning models. This includes unsupervised learning, deep reinforcement learning and so on. However, choosing the most appropriate models that could be used in games is difficult. This is because games produce incomplete data in real-time and may never produce a full dataset. Additionally, games generally require substantial computational resources to simulate physics and graphics. Heavy use of these models could result in a significant decrease in game performance and negatively impact players' experiences. As such, the models are generally selected based on these groups:

* Ability to perform online or incremental training.
* Ability to perform distributed training
* Computational complexity
* Sample efficiency
* Use cases

Ability to perform online or incremental training.

* Gradient-Based Methods
	+ Includes deep reinforcement learning
* Sufficient Statistics

Ability to perform distributed training

* Gradient-Based Methods

**Design Principles**

* Put practicability and flexibility first, speed second.
	+ Every API design choice made in this library is carefully analysed to ensure there is a balance between ease of use and control. This includes deciding which classes should be inherited and which should be composited.
	+ Some decisions, like making certain formulations as objects, allow us to satisfy both priorities at the same time. For example, treating regularizers as objects allows us to control computational resources by making it optional for our models. Making these formulations inbuilt to our model may increase the computational time due to redundant calculations as well as reducing flexibility, as these formulations cannot be recycled to other models.
* Simplicity over exhaustiveness
	+ Instead of implementing every variant of an algorithm, only the ones with a clear impact for real workloads are prioritised. For example, dimensionality reduction algorithms are not prioritised due to the use of high computational resources that could interfere with games, and practitioners can manually select features in games.
* Be minimal yet descriptive.
	+ Despite the simplicity on the surface, each line of library functions contains very descriptive code to reduce confusion.
* Easy to learn, hard to master.
	+ Related to the third principle, any new users who are learning this library will find this library intuitive, as they are only required to start from calling new(), train() and predict().
	+ These users can then gradually learn more advanced use cases without a huge leap of knowledge. For example, setting a regularizer object and appending it to the models’ new() constructor.

**General API Design**

DataPredict heavily relies on object-oriented programming architecture and uses both inheritance and compositional structure. This allows the sharing of extensions and utilities between different models, as well as having different configurations of these extensions and utilities by a single model. This flexibility allows the researchers and practitioners to adapt these models to their desired environments.

For example, DataPredict offers a way of having multiple deep reinforcement learning agents in different environments while having the same model using CategoricalPolicy and DiagonalGaussian quick setup objects. These allow the storing of individual environment states, actions and rewards for each agent without requiring this information to be stored in the deep reinforcement learning models themselves. This flexibility is further extended by having the deep reinforcement learning models act as a wrapper for a neural network model in the form of composition. This allows a single neural network to be shared with multiple deep reinforcement learning models without requiring the researchers and practitioners to write boilerplate code to access these deep reinforcement learning models’ functionalities. These deep reinforcement learning models mainly extend the neural network model’s update functions through the deep reinforcement learning models’ categoricalUpdate, diagonalGaussianUpdate and episodeUpdate functions that will be used by the CategoricalPolicy and DiagonalGaussian quick setup objects.

In addition, the library took extra precautions in performing inheritance. We ensured that when a class is inherited, we are confident that all modifications made in the extended class must affect the configuration of the inherited class. Another criterion for inheritance is that we want to ensure that the users of this library write minimal boilerplate code. One such example can be seen from the NeuralNetwork class, where it inherits from the IterativeMethodBaseModel class which in turn inherits from BaseModel class. In here, the BaseModel is responsible for handling model parameters initialisation and cost tracking. Meanwhile, the IterativeMethodBaseModel is responsible for tracking the number of iterations. Because the BaseModel’s cost tracking relies on IterativeMethodBaseModel, the IterativeMethodBaseModel must inherit the BaseModel as there is only one configuration possible from the BaseModel. As for the second criterion, one could agree that performing composition just to use NeuralNetwork model’s functionalities can end up producing too much boilerplate code without any clear benefits.

**Training Pipeline & Data Handling**

To reduce the learning curve for this library, we have imposed several rules that are built into our API.

Feature & Label Matrix

* By default, API assumes that the feature and label matrix are stored as a table of table of values.

Label Vector To Label Matrix Conversion

* For binary classification tasks, it is expected that the table contains tables of value with a length of 1.
* For multi-class classification tasks, our API automatically transform this to table containing tables of numerical values with a length equal to the number of classes. In addition, the label vector can take on non-numerical values, as these will be converted to a label matrix with numerical values. Users can also opt for using their own label matrix for more precise training.

Automatic Class Detection

* If the users do not set the classes before the training, the model will gather and store all the potential classes. This would then be used by the automatic label matrix conversion before the model can undergo training using the given dataset.

Model Parameters Generation

* Models will automatically generate model parameters based on the feature matrix and the classes that are given to the model. As such, users are not required to set the model parameters manually before training. Although the users can manually set the model parameters, we only expected this to be a use case of manually loading trained model parameters into the models.

**Additional Features**

Regularisation As An Object, Not A Formula.

* Given that many of the models use regularisation as part of their calculations, such as linear regression, logistic regression and neural networks, we designed the API in a way that these are additional options that could be inserted into the models. This way, we reduce the need for multiple models that only have regularisation in their differences, increasing the codebase maintainability.
* This approach differs from scikit-learn, where scikit-learn approaches this with variants of a model with different in-built regularizations.

Eligibility Traces As An Object, Not A Formula

* Just like the regularisation, eligibility traces are used by temporal difference methods of reinforcement learning models. As such, we made them objects to reduce redundant models, as these can be shared between different models.

Per Layer Learning Rate, Optimisers And Regularisation For Neural Networks

* In PyTorch and TensorFlow, the models generally use a single learning rate as well as a single optimiser for the whole model. They also do not support the classical weight regularisation that could provide an additional means to avoid overfitting.
* We find such limitations too restrictive, as users may want to perform various configurations, given how much freedom the game environment gives when interacting with these neural networks.

**Potential Use Cases**

Retention System

* Games provide many interactions between players and their environments, leading to a significant pool of data that could be exploited. Although studies related to dynamic difficulty adjustments have already been done, there is a lack of studies that exploit machine learning and deep reinforcement learning models to use these underutilised data that could be used to improve players’ retention.

Dynamic Difficulty Adjustment System

* Just like the use case stated above, games also provide many environmental interactions that could be exploited by the machine learning and deep reinforcement learning models that could be used to improve players’ retention.

In-Game Recommender System

* Players’ interactions with shops’ graphic user interface offer many opportunities for data collection. However, much of the research literature lacked this insight, which in turn led to a lack of studies related to in-game recommendation systems.

Targeting System

* In games, the model is not required to capture images to determine the location of the targets. The game engine already provides the precise locations of these targets, making the image inputs redundant. Additionally, because the location of the targets is already precise, it is possible to perform clustering tasks to find the centre of a targeted group.

Benchmarks

Machine Learning Models

Training

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | 1 Data | 10 Data | 100 Data | 1000 Data |
| Linear Regression |  |  |  |  |
| Logistic Regression |  |  |  |  |
| Neural Network (2 Layer + Manual Class Definition) |  |  |  |  |
| Neural Network (2 Layer + Automatic Class Definition) |  |  |  |  |
| K-Means (batch) |  |  |  |  |
| K-Means (Sequential) |  |  |  |  |

Inference

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | 1 Data | 10 Data | 100 Data | 1000 Data |
| Linear Regression |  |  |  |  |
| Logistic Regression |  |  |  |  |
| Neural Network (2 Layer + Manual Class Definition) |  |  |  |  |
| Neural Network (2 Layer + Automatic Class Definition) |  |  |  |  |
| K-Means (batch) |  |  |  |  |
| K-Means (Sequential) |  |  |  |  |

Deep Reinforcement learning

Deep Q-Learning Stats For Categorical update

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Extensions | 1 Experience | 10 Experiences | 100 Experiences | 1000 Experiences |
| None |  |  |  |  |
| Eligibility Traces |  |  |  |  |
| Uniform Experience Replay  |  |  |  |  |
| Eligibility Traces + Uniform Experience Replay |  |  |  |  |