Member-only story

Reinforcement Learning-Driven Adaptive Model Selection and Blending for Supervised Learning

Inspired by Deepseeker: Dynamically Choosing and Combining ML Models for Optimal Performance

Shenggang Li · Following

Published in Towards AI · 17 min read · 23 hours ago

176 3

Photo by Agence Olloweb on Unsplash

Introduction

Machine learning model selection has always been a challenge. Whether you're predicting stock prices, diagnosing diseases, or optimizing marketing campaigns, the question remains: which model works best for my data? Traditionally, we rely on cross-validation to test multiple models — *XGBoost*, *LGBM*, Random Forest, etc. — and pick the best one based on validation performance. But what if different parts of the dataset require different models? Or what if blending multiple models dynamically could improve accuracy?

This idea hit me while reading about Deepseeker R1, an advanced large language model (*LLM*) that adapts dynamically to improve performance. Inspired by its reinforcement learning (*RL*)-based optimization, I wondered:

can we apply a similar *RL*-driven strategy to supervised learning? Instead of manually selecting a model, why not let reinforcement learning learn the best strategy for us?

Imagine an RL agent acting like a data scientist — analyzing dataset characteristics, testing different models, and learning which performs best. Even better, rather than just picking one model, it could blend models dynamically based on data patterns. For instance, in a financial dataset, *XGBoost* might handle structured trends well, while *LGBM* might capture interactions better. Our *RL* system could switch between them intelligently or even combine them adaptively.

This paper proposes a novel Reinforcement Learning-Driven Model Selection and Blending framework. We frame the problem as a Markov Decision Process (*MDP*), where:

- The state represents dataset characteristics.
- The action is selecting or blending different *ML* models.
- The reward is based on model performance.
- The policy is trained using *RL* to find the best model selection strategy over time.

Unlike traditional approaches that apply a single best model across an entire dataset, this *RL*-driven method learns to choose the best model per data segment, or even blend models dynamically. This approach can automate, optimize, and personalize machine learning pipelines — reducing human intervention while improving predictive performance.

By the end of this paper, we'll see how reinforcement learning can transform model selection, making it adaptive, intelligent, and more efficient — just like a skilled data scientist who constantly learns and refines their choices.

Methodology: Reinforcement Learning for Adaptive Model Selection in Supervised Learning

I will describe the adaptive selection and blending of machine learning models as a Markov Decision Process (*MDP*) defined by the tuple *(S, A, P, R, γ)*, where:

- *S* is the set of states, representing the current statistical summary of the dataset (e.g., mean and variance of features),
- *A* is the set of actions, corresponding to selecting individual models *{XGB, LGBM, RF, DNN, Blend}*,
- *P(s′ | s, a)* defines the transition probabilities from the current state sss to the next state *s′*,
- *R(s, a)* is the immediate reward received after taking action *a* in state sss,
- $\gamma \in [0,1]$ is the discount factor that weighs immediate rewards versus future.

The goal of the reinforcement learning agent is to learn an optimal policy $\pi^*(s)$ that maximizes the expected cumulative reward over time:

$$\pi^*(s) = \arg\max_{\pi} \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\Big]$$

The reward *R(s, a)* at time *t* is defined as:

$$R(s_t, a_t) = \text{AUC}(a_t) + \text{KS}(a_t) - \text{Penalty}(a_t)$$

Where *AUC* and *KS* evaluate model prediction performance, and the penalty function balances model complexity (e.g., a higher penalty is applied to complex models like *DNN* or blending strategies).

State Representation and Actions

In many cases, the state sss is defined by feature-level summaries of the dataset, such as:

$$s = [\mu(X), \sigma^2(X)]$$

where *μ(X)* are the mean and variance vectors of the feature set *X*. Actions *a_t* can either select an individual model (e.g., *XGB*) or a weighted blend of multiple model predictions:

$$\hat{y}_{\text{Blend}} = \sum_{i=1}^{N} w_i \hat{y}_i$$

where *w_i* are blending weights, and *y^i* are predicted probabilities from each model.

Q-Learning and Model Evaluation

The heart of the reinforcement learning approach in this paper lies in solving the optimization problem of estimating the state-action value function *Q(s, a)*, which represents the expected cumulative reward starting from state sss, taking action *a*, and following a policy π. Here, the "state" sss is not just a generic concept — it encodes meaningful information about the data at hand, such as the statistical properties (mean, variance) of features, while the "action" *a* corresponds to selecting a specific model (e.g., *XGBoost*, *LightGBM*, RandomForest, *DNN*, or Blend) to make predictions.

The function *Q(s, a)* can be written as:

$$Q(s,a) = \mathbb{E}\left[R(s,a) + \gamma \max_{a'} Q(s',a')\right]$$

, where *R(s, a)* is the immediate reward from the model (based on *AUC + KS — Penalty*) after selecting action *a*, and γ is the discount factor that determines how much importance is given to future rewards. The goal of reinforcement learning is to maximize this cumulative reward by intelligently selecting models over time, and adapting as the data evolves.

Multi-Armed Bandit Method: Quick Model Selection Without Complexity

The multi-armed bandit approach treats the problem as stateless and memoryless, where rewards are independent across episodes. In the context of this paper, each action *a* is one of the candidate models (e.g., *XGBoost* or

LightGBM) or a blend of models. When action *a* is taken, we observe an immediate reward *R(a)* based on how well the selected model performs on metrics such as *AUC* and *KS*.

The update rule for the *Q-value* is

$$Q(a) \leftarrow Q(a) + \alpha\big(R(a) - Q(a)\big)$$

where,

$$\alpha = \frac{1}{N(a)}$$

is the learning rate, and *N(a)* is inversely proportional to the number of times the model *a* has been selected? The epsilon-greedy policy ensures a balance between exploring less-used models and exploiting the current best-performing one:

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(a) & \text{with probability } 1 - \epsilon \end{cases}$$

For example, in the earlier episodes, the bandit might explore *DNN* or blended models, once it consistently observes *XGBoost* achieving superior rewards (e.g., a high *AUC* with minimal penalties), it will converge toward

frequent exploitation of *XGBoost*. This approach works well when model performance does not depend heavily on the state of the data.

Deep Q-Network (DQN) Approach: When State Transitions Matter

Unlike the multi-armed bandit, the *DQN* approach generalizes the problem by considering state transitions and long-term rewards. In this paper, the state sss is defined by the statistical properties of the dataset (e.g., mean and variance of the features), which can change over time due to shifting data distributions. The agent's task is to choose the optimal model for the current state sss while anticipating how future states and rewards may evolve.

Q-values are approximated using a deep neural network $Q(s, a; \theta)$, where θ represents the network parameters. The update is performed by minimizing the temporal difference (*TD*) error:

$$\mathcal{L}(\theta) = \left[R(s,a) + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right]^2$$

Training the *DQN* involves experience replay, where past state-action-reward transitions *(s, a, r, s′)* are stored in a buffer and sampled randomly to stabilize training. This prevents overfitting to recent transitions and allows for better generalization across various states.

For example, suppose the current state *s* reflects a noisy dataset with high feature correlations (multicollinearity), this generally makes simple models like RandomForest ineffective. The *DQN* learns to avoid selecting RandomForest in this state and instead prioritize blending *XGBoost* and DNN predictions, which generalize better under noisy conditions. As the state

evolves — e.g., when noise levels decrease — the *DQN* dynamically updates its policy to exploit simpler models again.

Comparison: Exploration vs. Exploitation in Model Selection

Both approaches rely on the epsilon-greedy policy to explore models selected less frequently while exploiting the currently known best-performing model. However, the *DQN* approach introduces a more sophisticated exploration mechanism through the generalization power of neural networks. In dynamic scenarios where feature distributions change over time (e.g., in streaming financial datasets), the *DQN* learns to identify patterns across states and adjusts its model selection strategy accordingly.

Example Illustration: Choosing Between XGBoost and Blended Models

Consider an evolving financial dataset where the task is to predict loan defaults. In the initial state, where features exhibit strong linear separability, *XGBoost* may dominate due to its ability to handle tabular data efficiently. As the dataset transitions to a noisier state with overlapping features, a blended model combining *XGBoost* and *DNN* predictions may yield higher rewards due to its robustness. The agent continuously monitors the mean *AUC* and *KS* metrics from past decisions and updates its *Q-values* to reflect these changing conditions.

- In the Multi-Armed Bandit: The agent may converge to frequent exploitation of *XGBoost* based on early successes, missing opportunities when blended models perform better in noisy states.
- In the *DQN*: The agent dynamically adjusts its selection by tracking state transitions, selecting blended models when noise increases, and reverting to XGBoost during clean data periods.

Combining Both Approaches

The multi-armed bandit method provides a computationally efficient solution when rewards are independent and immediate, making it suitable for online model selection tasks.

Meanwhile, the *DQN* approach excels in dynamic environments where rewards depend on evolving data distributions, such as time-series forecasting or financial modeling.

Combining these approaches, this paper offers a robust framework for adaptive model selection and blending across a wide range of supervised learning tasks.

Data and Code Experiment

We begin by testing the Multi-Armed Bandit Method using the provided data and code, evaluating model selection performance through dynamic exploration and reward-based updates.

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import torch
import torch.nn as nn
import torch.nn.functional as F

# Read the dataset from CSV
data = pd.read_csv('rein_data_binary.csv')
X = data.drop('label', axis=1)
```

```
y = data['label']

# ---------------------------------------------
# 2) Metrics (AUC, KS)
# ---------------------------------------------
def calc_ks_score(y_true, y_prob):
    data = pd.DataFrame({'y_true': y_true, 'y_prob': y_prob}).sort_values('y_pro
    data['cum_pos'] = (data['y_true'] == 1).cumsum()
    data['cum_neg'] = (data['y_true'] == 0).cumsum()
    total_pos = data['y_true'].sum()
    total_neg = (data['y_true'] == 0).sum()
    data['cum_pos_rate'] = data['cum_pos'] / total_pos
    data['cum_neg_rate'] = data['cum_neg'] / total_neg
    data['ks'] = data['cum_pos_rate'] - data['cum_neg_rate']
    return data['ks'].max()

# ---------------------------------------------
# 3) PyTorch DNN Model
# ---------------------------------------------
class DNNModel(nn.Module):
    def __init__(self, input_dim=5):
        super(DNNModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, 16)
        self.fc2 = nn.Linear(16, 8)
        self.out = nn.Linear(8, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = torch.sigmoid(self.out(x))
        return x

def train_eval_pytorch_dnn(X_train, y_train, X_val, y_val,
                           epochs=5, batch_size=64, lr=1e-3, device='cpu'):
    model = DNNModel(input_dim=X_train.shape[1]).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.BCELoss()

    X_train_t = torch.tensor(X_train, dtype=torch.float32).to(device)
    y_train_t = torch.tensor(y_train, dtype=torch.float32).view(-1, 1).to(device
    X_val_t = torch.tensor(X_val, dtype=torch.float32).to(device)

    dataset_size = len(X_train_t)
    n_batches = (dataset_size // batch_size) + 1

    for epoch in range(epochs):
        indices = torch.randperm(dataset_size)
        X_train_t = X_train_t[indices]
        y_train_t = y_train_t[indices]
```

```
            for i in range(n_batches):
                start_idx = i * batch_size
                end_idx = start_idx + batch_size
                if start_idx >= dataset_size:
                    break

                x_batch = X_train_t[start_idx:end_idx]
                y_batch = y_train_t[start_idx:end_idx]

                preds = model(x_batch)
                loss = criterion(preds, y_batch)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        with torch.no_grad():
            val_preds = model(X_val_t).cpu().numpy().ravel()

        auc = roc_auc_score(y_val, val_preds)
        ks = calc_ks_score(y_val, val_preds)
        return model, auc, ks, val_preds

    # ------------------------------------------------
    # 4) Helper: Train & Evaluate Various Models
    # ------------------------------------------------
    def train_eval_model(model_name, X_train, y_train, X_val, y_val, device='cpu'):
        if model_name == 'xgb':
            model = XGBClassifier(use_label_encoder=False, eval_metric='logloss')
            model.fit(X_train, y_train)
            y_prob = model.predict_proba(X_val)[:, 1]
            auc = roc_auc_score(y_val, y_prob)
            ks = calc_ks_score(y_val, y_prob)
            return model, auc, ks, y_prob

        elif model_name == 'lgbm':
            model = LGBMClassifier()
            model.fit(X_train, y_train)
            y_prob = model.predict_proba(X_val)[:, 1]
            auc = roc_auc_score(y_val, y_prob)
            ks = calc_ks_score(y_val, y_prob)
            return model, auc, ks, y_prob

        elif model_name == 'rf':
            model = RandomForestClassifier()
            model.fit(X_train, y_train)
            y_prob = model.predict_proba(X_val)[:, 1]
            auc = roc_auc_score(y_val, y_prob)
            ks = calc_ks_score(y_val, y_prob)
            return model, auc, ks, y_prob
```

```
    elif model_name == 'dnn':
        model, auc, ks, y_prob = train_eval_pytorch_dnn(
            X_train.values, y_train.values, X_val.values, y_val.values, device=d
        )
        return model, auc, ks, y_prob

    else:
        raise ValueError(f"Unknown model name: {model_name}")

# Continue with the rest of the code unchanged...
# ------------------------------------------
# 5) Weighted Blending
# ------------------------------------------
def blend_predictions(probs_list, weights=None):
    if weights is None:
        weights = [1.0 / len(probs_list)] * len(probs_list)
    final_prob = np.zeros_like(probs_list[0])
    for w, p in zip(weights, probs_list):
        final_prob += w * p
    return final_prob

def evaluate_action(action, X_train, X_val, y_train, y_val, device='cpu'):
    """
    action: int from 0..4 => (xgb=0, lgbm=1, rf=2, dnn=3, blend=4)
    Returns:
      reward = (auc + ks) - penalty
      auc, ks
    """
    model_names = ['xgb', 'lgbm', 'rf', 'dnn']
    if action < 4:
        chosen_model = model_names[action]
        _, auc_val, ks_val, _ = train_eval_model(chosen_model, X_train, y_train,
        penalty = 0.05 if chosen_model == 'dnn' else 0.0
        reward = (auc_val + ks_val) - penalty
        return reward, auc_val, ks_val
    else:
        # Blend
        probs_list = []
        for m in model_names:
            _, auc_m, ks_m, p = train_eval_model(m, X_train, y_train, X_val, y_v
            probs_list.append(p)
        final_prob = blend_predictions(probs_list)
        auc_blend = roc_auc_score(y_val, final_prob)
        ks_blend = calc_ks_score(y_val, final_prob)
        reward = (auc_blend + ks_blend) - 0.1
        return reward, auc_blend, ks_blend

# ------------------------------------------
# 6) A Simple Multi-Armed Bandit Approach
```

```
# ------------------------------------------
def multi_armed_bandit_model_selection(
    n_episodes=50,
    n_actions=5,
    epsilon=0.06,
    device='cpu'
):
    """
    We have 5 actions (xgb=0, lgbm=1, rf=2, dnn=3, blend=4).
    For each 'episode':
      1) Generate a dataset (X,y) with the chosen seed
      2) Split into train/val
      3) Epsilon-greedy select an action
      4) Evaluate the chosen action => get reward
      5) Update average reward (Q) for that action
    """
    Q = np.zeros(n_actions, dtype=np.float32)
    counts = np.zeros(n_actions, dtype=int)

    # For storing raw AUC, KS, Reward each time an action is chosen
    action_auc_records = [[] for _ in range(n_actions)]
    action_ks_records = [[] for _ in range(n_actions)]
    action_reward_records = [[] for _ in range(n_actions)]

    action_history = []
    reward_history = []

    for episode in range(n_episodes):
        # Generate the data here
        seed = 1000 + episode
        X = data.drop('label', axis=1)  # Features
        y = data['label']  # Labels

        # Split the data
        X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, r

        # Epsilon-greedy action selection
        if np.random.rand() < epsilon:
            action = np.random.randint(n_actions)
        else:
            action = np.argmax(Q)

        # Evaluate chosen action => get (reward, auc, ks)
        reward, auc_val, ks_val = evaluate_action(
            action, X_train, X_val, y_train, y_val, device=device
        )

        # Update Q (incremental mean)
        counts[action] += 1
        Q[action] += (reward - Q[action]) / counts[action]
```

```
        # Store details
        action_history.append(action)
        reward_history.append(reward)
        action_auc_records[action].append(auc_val)
        action_ks_records[action].append(ks_val)
        action_reward_records[action].append(reward)

        print(f"Episode {episode+1}/{n_episodes}, "
              f"Action={action}, Reward={reward:.4f}, Updated Q={Q}")

    return Q, action_history, reward_history, action_auc_records, action_ks_reco

# ---------------------------------------------
# 7) Run the Bandit, then Interpret Results
# ---------------------------------------------
def run_bandit():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print(f"Using device={device}")

    n_episodes = 50
    n_actions = 5
    epsilon = 0.05

    (
        Q,
        actions,
        rewards,
        auc_records,
        ks_records,
        reward_records
    ) = multi_armed_bandit_model_selection(
        n_episodes=n_episodes,
        n_actions=n_actions,
        epsilon=epsilon,
        device=device
    )

    best_action = np.argmax(Q)
    model_names = ["XGB", "LightGBM", "RandomForest", "DNN", "Blend"]

    print("\n=======================================")
    print("Interpreting Your Current Results")
    print("=======================================\n")

    print("Final Q-values:", Q)
    print(f"Best action index: {best_action}")
    print(f"Best action is: {model_names[best_action]} with estimated Q = {Q[bes

    print("Detailed AUC/KS/Reward by action:")
```

```
        print("-------------------------------------------------")
        for a in range(n_actions):
            if len(auc_records[a]) > 0:
                avg_auc = np.mean(auc_records[a])
                avg_ks = np.mean(ks_records[a])
                avg_reward = np.mean(reward_records[a])
                print(f"Action {a} ({model_names[a]}): chosen {len(auc_records[a])}
                print(f"  Mean AUC = {avg_auc:.4f}, Mean KS = {avg_ks:.4f}, Mean Rew
            else:
                print(f"Action {a} ({model_names[a]}): chosen 0 times\n")


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

Interpretation of Results

The final *Q-values* indicate the estimated performance of each model in terms of cumulative rewards. The results from the experiment reveal:

- Best action: *XGBoost* with an average reward of *1.267*.
- Comparison: *XGBoost* significantly outperformed other models due to its balance of prediction accuracy and robustness.

The detailed results provide insight into the trade-offs between different models:

Model	Chosen Times	Mean AUC	Mean KS	Mean Reward
XGBoost	49	0.7881	0.4791	1.2671
LightGBM	0	-	-	-
Random Forest	0	-	-	-
DNN	1	0.7695	0.4852	1.2047
Blend	0	-	-	-

The results suggest that:

- *XGBoost* dominates the selection because of its consistently high *AUC* and *KS* scores.
- *DNN* shows potential, as indicated by its relatively high *AUC* and *KS* in the one episode it was selected.
- Blend and other models were not frequently chosen, suggesting further experimentation with weights and feature representations might be needed.

The dominance of *XGBoost* highlights its suitability for this particular data structure and reward mechanism. Further research could include:

1. Exploring dynamic penalty adjustment for complex models.
2. Extending the bandit framework to handle time-varying rewards.
3. Optimizing model blending strategies using reinforcement learning itself.

It shows that the Multi-Armed Bandit Method can efficiently find the best model based on the given evaluation criteria.

Next, let's dive into testing the Deep Q-Network (*DQN*) approach. This will help us see how well it handles changing states and long-term rewards, compared to the simpler, stateless Multi-Armed Bandit Method we tested earlier.

```
import numpy as np
import pandas as pd
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import torch
import torch.nn as nn
import torch.nn.functional as F

# Gymnasium
import gymnasium as gym
from gymnasium import spaces

# Stable Baselines3
from stable_baselines3 import DQN
from stable_baselines3.common.vec_env import DummyVecEnv

# For callback
from stable_baselines3.common.callbacks import BaseCallback

# ---------------------------------------------
# 1) Read data from CSV file
# ---------------------------------------------
data = pd.read_csv('rein_data_binary.csv')
X = data.drop('label', axis=1)  # Features
y = data['label']  # Labels

# ---------------------------------------------
# 2) Metrics (AUC, KS)
# ---------------------------------------------
def calc_ks_score(y_true, y_prob):
    data = pd.DataFrame({'y_true': y_true, 'y_prob': y_prob}).sort_values('y_pro
    data['cum_pos'] = (data['y_true'] == 1).cumsum()
    data['cum_neg'] = (data['y_true'] == 0).cumsum()
    total_pos = data['y_true'].sum()
    total_neg = (data['y_true'] == 0).sum()
    data['cum_pos_rate'] = data['cum_pos'] / total_pos
    data['cum_neg_rate'] = data['cum_neg'] / total_neg
    data['ks'] = data['cum_pos_rate'] - data['cum_neg_rate']
    return data['ks'].max()

# ---------------------------------------------
# 3) PyTorch DNN
# ---------------------------------------------
class DNNModel(nn.Module):
    def __init__(self, input_dim=5):
        super(DNNModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, 16)
        self.fc2 = nn.Linear(16, 8)
        self.out = nn.Linear(8, 1)
```

```
        def forward(self, x):
            x = F.relu(self.fc1(x))
            x = F.relu(self.fc2(x))
            x = torch.sigmoid(self.out(x))
            return x

    def train_eval_pytorch_dnn(X_train, y_train, X_val, y_val,
                               epochs=5, batch_size=64, lr=1e-3, device='cpu'):
        model = DNNModel(input_dim=X_train.shape[1]).to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=lr)
        criterion = nn.BCELoss()

        X_train_t = torch.tensor(X_train, dtype=torch.float32).to(device)
        y_train_t = torch.tensor(y_train, dtype=torch.float32).view(-1, 1).to(device
        X_val_t = torch.tensor(X_val, dtype=torch.float32).to(device)

        dataset_size = len(X_train_t)
        n_batches = (dataset_size // batch_size) + 1

        for epoch in range(epochs):
            indices = torch.randperm(dataset_size)
            X_train_t = X_train_t[indices]
            y_train_t = y_train_t[indices]

            for i in range(n_batches):
                start_idx = i * batch_size
                end_idx = start_idx + batch_size
                if start_idx >= dataset_size:
                    break

                x_batch = X_train_t[start_idx:end_idx]
                y_batch = y_train_t[start_idx:end_idx]

                preds = model(x_batch)
                loss = criterion(preds, y_batch)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        with torch.no_grad():
            val_preds = model(X_val_t).cpu().numpy().ravel()

        auc = roc_auc_score(y_val, val_preds)
        ks = calc_ks_score(y_val, val_preds)
        return model, auc, ks, val_preds

    # ------------------------------------------
    # 4) Train & Evaluate Helper
```

```
# ---------------------------------------------
def train_eval_model(model_name, X_train, y_train, X_val, y_val, device='cpu'):
    if model_name == 'xgb':
        model = XGBClassifier(use_label_encoder=False, eval_metric='logloss')
        model.fit(X_train, y_train)
        y_prob = model.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_prob)
        ks = calc_ks_score(y_val, y_prob)
        return model, auc, ks, y_prob

    elif model_name == 'lgbm':
        model = LGBMClassifier()
        model.fit(X_train, y_train)
        y_prob = model.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_prob)
        ks = calc_ks_score(y_val, y_prob)
        return model, auc, ks, y_prob

    elif model_name == 'rf':
        model = RandomForestClassifier()
        model.fit(X_train, y_train)
        y_prob = model.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_prob)
        ks = calc_ks_score(y_val, y_prob)
        return model, auc, ks, y_prob

    elif model_name == 'dnn':
        model, auc, ks, y_prob = train_eval_pytorch_dnn(
            X_train.values, y_train.values, X_val.values, y_val.values, device=d
        )
        return model, auc, ks, y_prob

    else:
        raise ValueError(f"Unknown model name: {model_name}")

def blend_predictions(probs_list, weights=None):
    if weights is None:
        weights = [1.0 / len(probs_list)] * len(probs_list)
    final_prob = np.zeros_like(probs_list[0])
    for w, p in zip(weights, probs_list):
        final_prob += w * p
    return final_prob

# ---------------------------------------------
# 5) Single-step Environment
# ---------------------------------------------
class ModelSelectionEnv(gym.Env):
    metadata = {"render_modes": ["human"]}

    def __init__(self, X, y, device='cpu'):
```

```
        super().__init__()
        self.device = device

        # Train/val split
        self.X_train, self.X_val, self.y_train, self.y_val = train_test_split(
            X, y, test_size=0.3, random_state=123
        )

        means = X.mean().values
        vars_ = X.var().values
        self.state = np.concatenate([means, vars_])  # observation

        # 5 discrete actions
        self.action_space = spaces.Discrete(5)
        self.observation_space = spaces.Box(
            low=-np.inf,
            high=np.inf,
            shape=(len(self.state),),
            dtype=np.float32
        )
        self.terminated = False

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.terminated = False
        return self.state.astype(np.float32), {}

    def step(self, action):
        if self.terminated:
            return self.state.astype(np.float32), 0.0, True, False, {}

        model_names = ['xgb', 'lgbm', 'rf', 'dnn']
        if action < 4:
            chosen_model = model_names[action]
            _, auc_v, ks_v, _ = train_eval_model(
                chosen_model,
                self.X_train, self.y_train,
                self.X_val, self.y_val,
                device=self.device
            )
            penalty = 0.05 if chosen_model == 'dnn' else 0.0
            reward = (auc_v + ks_v) - penalty
            info = {
                "action_name": chosen_model,
                "AUC": auc_v,
                "KS": ks_v,
                "Penalty": penalty
            }
        else:
            # Blend
```

```
            probs_list = []
            for m in model_names:
                _, auc_m, ks_m, prob_m = train_eval_model(
                    m,
                    self.X_train, self.y_train,
                    self.X_val, self.y_val,
                    device=self.device
                )
                probs_list.append(prob_m)
            final_prob = blend_predictions(probs_list)
            auc_v = roc_auc_score(self.y_val, final_prob)
            ks_v = calc_ks_score(self.y_val, final_prob)
            penalty = 0.1
            reward = (auc_v + ks_v) - penalty
            info = {
                "action_name": "blend",
                "AUC": auc_v,
                "KS": ks_v,
                "Penalty": penalty
            }

        self.terminated = True
        return self.state.astype(np.float32), reward, True, False, info

# ---------------------------------------------
# 7) RL Training & Execution
# ---------------------------------------------
def run_rl_model_selection_pytorch():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print(f"Using {device} device")

    # Create single-step Gymnasium environment
    env = ModelSelectionEnv(X, y, device=device)

    # Wrap with DummyVecEnv
    def make_env():
        return env
    vec_env = DummyVecEnv([make_env])

    # Create callback
    callback = BanditSummaryCallback()

    # Create DQN
    model = DQN(
        "MlpPolicy",
        vec_env,
        verbose=1,
        learning_rate=1e-3,
        buffer_size=10000,
        exploration_fraction=0.3,
```

```
            exploration_final_eps=0.02,
            tensorboard_log="./rl_tensorboard/"
        )

        # Train with callback
        model.learn(total_timesteps=2000, callback=callback)

        # Evaluate final policy (one step)
        obs = vec_env.reset()
        action, _ = model.predict(obs, deterministic=True)
        obs, rewards, dones, infos = vec_env.step(action)
        final_reward = rewards[0]
        action_map = ["XGB", "LightGBM", "RandomForest", "DNN", "Blend"]
        print("\n====================================")
        print(f"Final chosen action => {action[0]} ({action_map[action[0]]})")
        print(f"Final step reward => (AUC + KS - penalty) = {final_reward:.4f}")
        print("====================================\n")

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

Results and Interpretation

Training Summary:

- *2000* episodes were run with exploration and exploitation driven by the epsilon-greedy policy.
- *Q-values* reflect the estimated cumulative rewards for each model selection.

Final Q-values:

[1.1442,1.1800,1.0847,1.0510,1.0684]

Best action: *LightGBM* with the highest *Q-value 1.1800*.

Detailed performance:

Model	Chosen Times	Mean AUC	Mean KS	Mean Reward
XGB	111	0.745	0.3993	1.1442
LightGBM	1673	0.7554	0.4247	1.18
RandomFor	101	0.7248	0.3599	1.0847
DNN	55	0.7228	0.3782	1.051
Blend	60	0.7536	0.4148	1.0684

Final chosen action:
LightGBM (action index 1) consistently outperformed others on *AUC, KS*, and reward. Given the data structure and evaluation metrics, its selection demonstrates robustness and reliability.

Summary of Deepseeker Techniques for Reinforcement Learning in Adaptive Model Selection

I wrote this paper by drawing inspiration from Deepseeker Techniques to improve adaptive model selection:

Reinforcement Learning without Supervised Fine-Tuning: Deepseeker skips the need for large labeled datasets by using reinforcement learning (*RL*) to develop reasoning through trial-and-error.

I apply the RL agent dynamically learns the best model selection policy through exploration and feedback, without fine-tuning. Over time, it generalizes across new datasets or domains without retraining models, mimicking Deepseeker's meta-learning. Example: The agent may learn to use XGBoost for structured data and *DNN* for non-linear data, automatically adapting to changing conditions.

Data-Driven Model Selection and Classification: Deepseeker activates only the relevant model components based on the data context, making the process efficient and targeted. Similarly, the *RL* agent uses dataset statistics (mean, variance, etc.) to classify data regions and match them with the best models.

For example, for structured regions, the agent may choose *XGBoost*, while noisy, non-linear regions could trigger blended *DNN* and *LGBM* models. This dynamic mapping of states to actions prevents applying a single model globally and improves prediction accuracy.

Partial Parameter Activation (Sparse Model Execution): Instead of activating all parameters, Deepseeker selectively activates only those needed for the current task.

The same concept applies to dynamic model blending, activating only the most effective models or configurations based on the current state. Example: In a medical dataset, *XGBoost* could be used for structured patient history, while *DNN* handles complex genetic data, minimizing unnecessary computation and boosting efficiency.

Chain-of-thought (CoT): DeepSeek-R1-Zero uses chain-of-thought (CoT) reasoning to solve complex tasks by generating self-verifying and reflective chains of reasoning steps.

In this paper, the RL agent could potentially use CoT for adaptive model selection (not applied yet, but it's a research direction worth exploring):

Open in app ↗

Medium Search Write 2

performance (*AUC, KS*). If rewards are low, it reflects, adjusts blending, or picks a new model. With rolling feedback, it remembers past decisions to avoid repeating mistakes and explores blending ratios dynamically for continuous improvement.

Final Thoughts

I have explored how reinforcement learning (*RL*) can power adaptive model selection and blending by treating it as a Markov Decision Process (*MDP*). The *RL* agent intelligently navigates choices like *XGBoost*, *LightGBM, DNN*, or model blending, dynamically adapting to shifting data distributions. Using cumulative performance rewards (based on *AUC* and *KS*) and penalizing inefficient choices, we can optimize the modeling process without relying on fixed assumptions.

I'm inspired by the reinforcement learning mechanism in Deepseeker NN (LLM), which adds domain-specific knowledge. This could help generalize across various tasks like automated trading, healthcare diagnostics, and predictive maintenance.

Challenges, such as computational overhead and exploration-exploitation balance, remain. However, future meta-learning and semi-supervised labeling could improve adaptability while reducing costs. Expanding to *RL*-based hyperparameter tuning could make the system fully autonomous, choosing the optimal model and its configurations.

Finally, this paper sets the stage for next-generation AutoML systems, where reinforcement learning and large language models collaborate for smarter,

scalable solutions in real-world applications.

The data and code can be accessed at https://github.com/datalev001/RL_supLR/.

About me

With over 20 years of experience in software and database management and 25 years teaching IT, math, and statistics, I am a Data Scientist with extensive expertise across multiple industries.

You can connect with me at:

Email: datalev@gmail.com | LinkedIn | X/Twitter

Reinforcement Learning | Deepseek | AI | Supervised Learning | Xgboost

Published in Towards AI

73K Followers · Last published just now

Follow

The leading AI community and content platform focused on making AI accessible to all. Check out our new course platform: https://academy.towardsai.net/courses/beginner-to-advanced-llm-dev

Written by Shenggang Li

1.7K Followers · 71 Following

Following

Responses (3)

What are your thoughts?

Respond

Jonathansholmes
8 hours ago

I am concerned about parameter optimisations as some models are sensitive to these given a data set. I would like to experiment and trial versions of the same model for example but with different parameters to begin with before looking at a perhaps... more

2 Reply

Derrick Johnson
6 hours ago

Thanks for sharing this great article!

3 Reply

S

Simon Horner
3 hours ago

Absolutely brilliant! Thank you

Reply

More from Shenggang Li and Towards AI

In Towards AI by Shenggang Li

Decoding Ponzi Schemes with Math and AI

Experiments in Action: Unveiling Hidden Trends in Stocks and Markets Using Mamba...

Jan 13 440 2

In Towards AI by Krishan Walia

Fine-tuning DeepSeek R1 to respond like Humans using Python!

Learn to Fine-Tune Deep Seek R1 to respond as humans, through this beginner-friendly...

2d ago 21

In Towards AI by Mohit Sewak, Ph.D.

Artificial Super Intelligence (ASI): The Research Frontiers to Achiev...

Examining the Latest Research Advancements and Their Implications for AS...

6d ago 62 1

In Towards AI by Shenggang Li

Graph Neural Networks: Unlocking the Power of Relationships in...

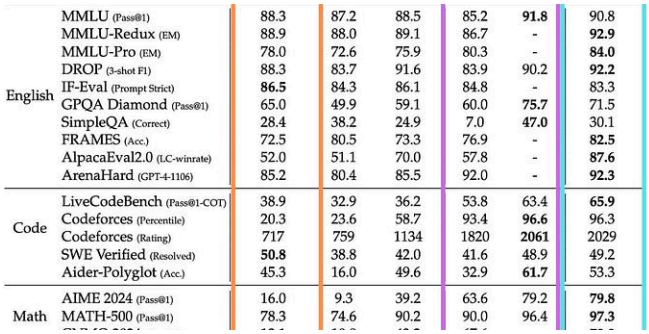
Exploring the Concepts, Types, and Real-World Applications of GNNs in Feature...

Jan 11 253 3

See all from Shenggang Li

See all from Towards AI

Recommended from Medium



English	MMLU (Pass@1)	88.3	87.2	88.5	85.2	**91.8**	90.8
	MMLU-Redux (EM)	88.9	88.0	89.1	86.7	-	**92.9**
	MMLU-Pro (EM)	78.0	72.6	75.9	80.3	-	**84.0**
	DROP (3-shot F1)	88.3	83.7	91.6	83.9	90.2	**92.2**
	IF-Eval (Prompt Strict)	**86.5**	84.3	86.1	84.8	-	83.3
	GPQA Diamond (Pass@1)	65.0	49.9	59.1	60.0	**75.7**	71.5
	SimpleQA (Correct)	28.4	38.2	24.9	7.0	**47.0**	30.1
	FRAMES (Acc.)	72.5	80.5	73.3	76.9	-	**82.5**
	AlpacaEval2.0 (LC-winrate)	52.0	51.1	70.0	57.8	-	**87.6**
	ArenaHard (GPT-4-1106)	85.2	80.4	85.5	92.0	-	**92.3**
Code	LiveCodeBench (Pass@1-COT)	38.9	32.9	36.2	53.8	63.4	**65.9**
	Codeforces (Percentile)	20.3	23.6	58.7	93.4	**96.6**	96.3
	Codeforces (Rating)	717	759	1134	1820	**2061**	2029
	SWE Verified (Resolved)	**50.8**	38.8	42.0	41.6	48.9	49.2
	Aider-Polyglot (Acc.)	45.3	16.0	49.6	32.9	**61.7**	53.3
Math	AIME 2024 (Pass@1)	16.0	9.3	39.2	63.6	79.2	**79.8**
	MATH-500 (Pass@1)	78.3	74.6	90.2	90.0	96.4	**97.3**

In Artificial Intelligence in Plain ... by BavalpreetSi...

DeepSeek R1: Understanding GRPO and Multi-Stage Training

Artificial intelligence has taken a significant leap forward with the release of DeepSeek R...

Jan 27 391

Isaak Kamau

A Simple Guide to DeepSeek R1: Architecture, Training, Local...

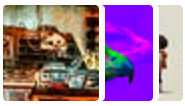
DeepSeek's Novel Approach to LLM Reasoning

Jan 23 3.3K 35

Lists

Generative AI Recommended Reading

52 stories · 1632 saves

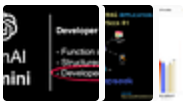


What is ChatGPT?

9 stories · 499 saves

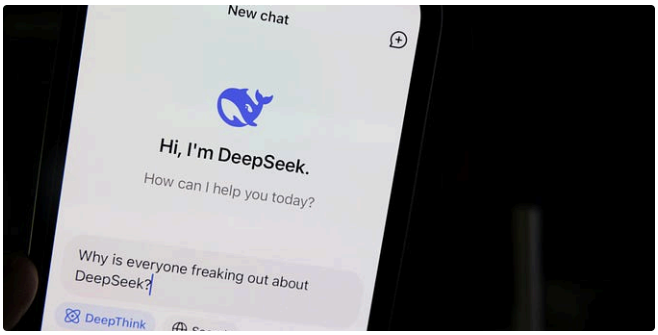
The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 547 saves



Natural Language Processing

1916 stories · 1571 saves



In Data Science in your pocket by Mehul Gupta

What is GRPO ? The RL algorithm used to train DeepSeek

Maths behind GRPO Reinforcement Learning algorithm leading to DeepSeek

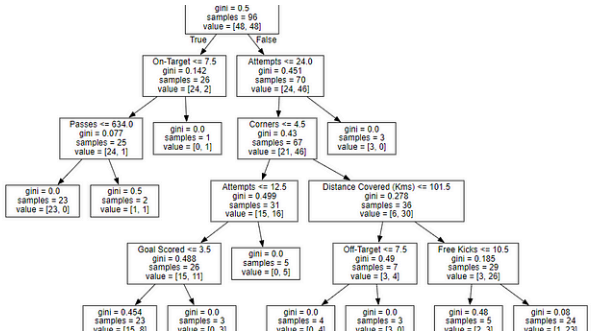
4d ago 83

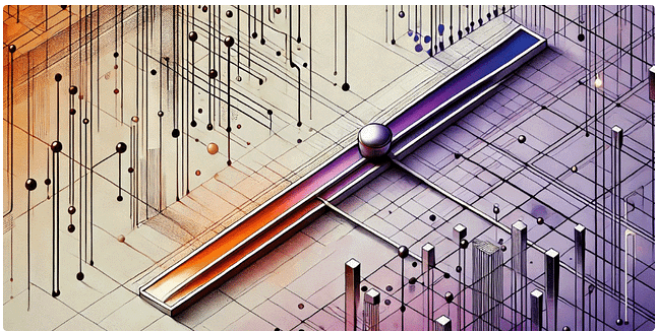
AI Papers Academy

DeepSeek-R1 Paper Explained—A New RL LLMs Era in AI?

Dive into the groundbreaking DeepSeek-R1 research paper, introduces open-source...

Jan 25 256 3





G.Thompson

Machine Learning Explainability

What Types of Insights Are Possible

6d ago 2

Nigel Gebodh

Why Does My LLM Have A Temperature?

Understanding the temperature parameter in LLMs

Jan 6 115 1

See more recommendations