Reinforced Learning in Asset Pricing

Jesus Villota Miranda[†]

⟨ [†]CEMFI, Calle Casado del Alisal, 5, 28014 Madrid, Spain ⟩

⟨ Email: jesus.villota@cemfi.edu.es ⟩

Abstract

JEL Codes:
Keywords:

Contents

Conversation w/ ChatGPT

USE BAYESIAN REINFORCEMENT LEARNING

Conversation w/ ChatGPT

1. Model

1.1 No-Arbitrage Asset Pricing

Similar to CPZ, our model is built on the fundamental concept of no-arbitrage in the cross-section of asset returns. Let $R_{t+1,i}$ denote the return of asset i at time $t+1$, and let M_{t+1} represent the stochastic discount factor (SDF). For any return in excess of the risk-free rate, denoted as $R^e_{t+1,i} = R_{t+1,i} - R^f_{t+1}$, the no-arbitrage condition implies:

$$\mathbb{E}_t \left[M_{t+1} R^e_{t+1,i} \right] = 0, \tag{1}$$

where $\mathbb{E}_t[\cdot]$ represents the expectation conditional on the information set at time t. The SDF is formulated as an affine combination of factors, where the goal is to use reinforcement learning to determine optimal SDF weights that minimize pricing errors.

1.2 Reinforcement Learning Framework

To apply reinforcement learning, we define a Markov Decision Process (MDP) to model the asset pricing problem:

- **State Space ($\mathcal{S}$)**: The state at time t, s_t, is represented by the information set I_t, which includes macroeconomic variables, firm-specific characteristics, and historical returns. Formally, $s_t = [I_t, I_{t,i}]$, where I_t represents macroeconomic variables, and $I_{t,i}$ are firm-specific characteristics.

- **Action Space ($\mathcal{A}$)**: The action a_t represents the choice of SDF portfolio weights, ω_t, at each time t. These weights determine the construction of the SDF.

- **Reward Function (R)**: The reward at time $t+1$, r_{t+1}, is defined as the negative of the squared pricing error:

$$r_{t+1} = -\left(M_{t+1} R^e_{t+1,i} \right)^2 = -\left(\left(1 - \omega_t^\top R^e_{t+1} \right) R^e_{t+1,i} \right)^2. \tag{2}$$

This reward function encourages the agent to minimize pricing errors, thereby finding SDF weights that best price the cross-section of returns.

- **Transition Dynamics** (P): The transition from state s_t to state s_{t+1} depends on the evolution of macroeconomic and firm-specific variables. These transitions are modeled based on the observed data.

The objective is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the expected cumulative reward over time:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_{t+1} \mid s_0, \pi\right], \tag{3}$$

where $\gamma \in [0, 1]$ is the discount factor, which ensures that the agent prioritizes immediate rewards while considering future pricing errors.

1.3 Actor-Critic Algorithm

To solve the optimization problem, we adopt an **Actor-Critic** approach, where the actor learns the policy and the critic estimates the value function.

- **Actor**: The actor is a neural network that takes the state s_t as input and outputs the portfolio weights ω_t. The actor is updated by using the **policy gradient** to adjust the weights in the direction that improves the cumulative reward.

- **Critic**: The critic evaluates the chosen actions by estimating the **value function** $V(s_t)$, which represents the expected cumulative reward starting from state s_t. The critic is updated by minimizing the **temporal difference (TD) error**:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t). \tag{4}$$

The actor and critic networks are both parameterized by feedforward neural networks combined with LSTMs, similar to the model architecture in CPZ, to capture the time-series dependencies in macroeconomic variables.

2. Estimation

2.1 Loss Function and Training Procedure

The empirical loss function for training the actor and critic is defined as the negative of the cumulative reward. For the critic, the loss function is the mean squared TD error:

$$L_{\text{critic}} = \frac{1}{T} \sum_{t=0}^{T} \delta_t^2. \tag{5}$$

For the actor, we use the **policy gradient theorem** to update the policy parameters θ:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot \delta_t\right]. \tag{6}$$

We train the actor and critic using **stochastic gradient descent (SGD)** with an adaptive learning rate. **Dropout** is employed as a regularization technique to prevent overfitting, similar to CPZ.

2.2 Hyperparameter Tuning and Ensemble Learning

We use an ensemble approach to achieve robust and stable estimates. The models are trained multiple times with different initializations, and the final portfolio weights are averaged over all model fits:

$$\hat{\omega} = \frac{1}{M} \sum_{m=1}^{M} \hat{\omega}^{(m)}, \tag{7}$$

where M is the number of ensemble models. We tune hyperparameters such as the number of layers, nodes per layer, discount factor γ, and learning rate using a validation set to maximize the Sharpe ratio of the SDF on the validation data.

3. Model Comparison

We compare our RL-based asset pricing model with the GAN and FFN models from CPZ. The comparison is based on three metrics:

1. **Sharpe Ratio (SR)**: The unconditional Sharpe ratio of the SDF portfolio, defined as:

$$SR = \frac{\mathbb{E}[F]}{\sqrt{\text{Var}[F]}}. \tag{8}$$

2. **Explained Variation (EV)**: The proportion of variation in asset returns explained by the SDF.

3. **Cross-Sectional R^2 (XS-R^2)**: The cross-sectional R^2 measure that captures the average pricing error normalized by the average mean return.

4. Conclusion

This paper introduces a novel application of Reinforcement Learning to asset pricing, building upon the methodology of Chen, Pelger, and Zhu. By modeling the asset pricing problem as a Markov Decision Process and using an Actor-Critic algorithm, we provide a framework that captures complex relationships in asset returns, while ensuring no-arbitrage pricing. Our results demonstrate that RL can achieve competitive pricing performance compared to state-of-the-art deep learning methods.

5. Methodology

5.1 Reinforcement Learning for Asset Pricing

Reinforcement Learning (RL) is a machine learning paradigm in which an agent interacts with an environment by taking actions, receiving rewards, and updating its policy to maximize long-term cumulative rewards. In the context of asset pricing, the agent represents the model that estimates the SDF and portfolio weights, while the environment is the financial market that provides information about returns, risk factors, and macroeconomic variables.

Formally, the RL framework is defined as a *Markov Decision Process (MDP)*, which consists of the following components:

5.1.1 State Space (s_t)

The *state space* s_t at time t contains all relevant information needed to price assets and estimate the SDF. In our framework, the state space is composed of:

$$s_t = (I_t, I_{t,i}, \omega_{t-1}, h_t)$$

where:

- I_t: A vector of macroeconomic variables at time t (e.g., GDP growth, inflation, interest rates).
- $I_{t,i}$: A vector of firm-specific characteristics at time t for asset i (e.g., size, book-to-market ratio).
- ω_{t-1}: The SDF portfolio weights from the previous time step.
- h_t: A set of hidden state variables that capture the dynamic patterns in the macroeconomic data, estimated via a Long Short-Term Memory (LSTM) network.

The inclusion of h_t allows the model to incorporate both short-term and long-term dependencies in the macroeconomic and firm-specific data. The hidden states h_t are updated at each time step as new macroeconomic information becomes available.

5.1.2 Action Space (a_t)

At each time step t, the agent selects an *action* a_t that adjusts the SDF portfolio weights ω_t and the risk exposures β_t. The action space can be either continuous or discrete, depending on the nature of the adjustments. We define the action space as continuous:

$$a_t = (\omega_t, \beta_t)$$

where:

- ω_t: The portfolio weights at time t for the SDF.
- β_t: The exposure to systematic risks at time t.

The goal of the agent is to choose actions that maximize the reward over time, which we define next.

5.1.3 Reward Function (R_t)

The *reward function* measures the performance of the agent at each time step and is used to guide the learning process. In our framework, the reward function reflects the trade-off between maximizing risk-adjusted returns and minimizing pricing errors. We define the reward function as:

$$R_t = SR(\omega_t) - \lambda \cdot PE(\omega_t, \beta_t)$$

where:

- $SR(\omega_t)$: The Sharpe ratio of the portfolio with weights ω_t, defined as:

$$SR(\omega_t) = \frac{\mathbb{E}[R_t]}{\sqrt{\text{Var}(R_t)}}$$

  where $R_t = \omega_t^\top R_{t+1}^e$ is the portfolio return based on the excess returns R_{t+1}^e of the assets.

- $PE(\omega_t, \beta_t)$: The pricing error, which measures the deviation of the model's predicted returns from observed returns, defined as:

$$PE(\omega_t, \beta_t) = \sum_{i=1}^{N} \left(R_{t+1,i}^e - \beta_{t,i} F_{t+1} \right)^2$$

- λ: A regularization parameter that balances the trade-off between maximizing the Sharpe ratio and minimizing the pricing error.

The reward function ensures that the agent learns to both optimize portfolio weights for high risk-adjusted returns and minimize the mispricing of assets.

5.1.4 Policy and Value Functions

The agent's objective is to learn a policy $\pi(a_t \mid s_t)$ that maps states to actions in a way that maximizes the expected cumulative reward over time:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t \mid \pi \right]$$

where $\gamma \in (0, 1]$ is a discount factor that determines the importance of future rewards. The policy π is typically parameterized by a neural network, and the optimization is done using a gradient-based method such as *policy gradient* or *Deep Q-Learning*.

The policy is updated iteratively as the agent interacts with the market environment. At each time step, the agent observes the current state s_t, selects an action a_t based on the current policy, and receives a reward R_t. The policy is then updated to improve the expected cumulative reward.

5.1.5 Actor-Critic Architecture

We adopt an *actor-critic* architecture to implement the RL model:

- The **actor** network represents the policy $\pi(s_t)$ and outputs the action $a_t = \{\omega_t, \beta_t\}$.

- The **critic** network estimates the value function $V^\pi(s_t)$ and evaluates the quality of the current state-action pair.

The critic network helps the actor update its policy by providing feedback on the long-term value of the actions taken. This structure allows for continuous improvement of the policy over time.

5.2 LSTM for Macroeconomic Dynamics

Macroeconomic variables are often non-stationary and exhibit complex dynamic patterns. To capture these dynamics, we use a *Long Short-Term Memory (LSTM)* network to estimate hidden state variables h_t that summarize the information in the macroeconomic time series. The LSTM processes the sequence of macroeconomic observations $(I_0, I_1, \ldots, I_t)$ and outputs a hidden state h_t that reflects both short-term and long-term dependencies.

Formally, the LSTM updates the hidden state according to the following equations:

$$h_t = \sigma\left(W_h h_{t-1} + W_x I_t + b_h\right)$$

where σ is a non-linear activation function, W_h and W_x are weight matrices, and b_h is a bias term. The hidden state h_t is then used as an input to the RL agent for making decisions about portfolio adjustments.

5.3 GAN and Reinforcement Learning Integration

In our model, we integrate the *Generative Adversarial Network (GAN)* from *Chen, Pelger, and Zhu (2023)* into the RL framework. The adversarial network in GAN selects the hardest-to-price moments, and the RL agent learns to adjust the portfolio weights and risk exposures to minimize the worst-case pricing errors. The GAN serves as the environment, continually presenting challenges for the RL agent to solve.

At each step, the RL agent interacts with the GAN by selecting portfolio weights ω_t and risk loadings β_t, and the GAN adversary responds by identifying the moments where pricing errors are largest. The RL agent then updates its policy based on the reward function, adjusting its actions to minimize these errors and maximize the Sharpe ratio.

6. Implementation

In this section, we describe the technical implementation of the Reinforcement Learning (RL) model for asset pricing. We outline the neural network architecture used for policy optimization, the training procedure, the regularization techniques employed to prevent overfitting, and the

hyperparameter tuning process. Additionally, we discuss the integration of Generative Adversarial Networks (GANs) with the RL framework to further enhance the model's ability to capture pricing errors in asset returns.

6.1 Neural Network Architecture

The RL model is implemented using a neural network architecture to parameterize both the policy and the value function in the actor-critic framework. The two main components of the architecture are the *actor network* and the *critic network*.

6.1.1 Actor Network

The *actor network* represents the policy $\pi(s_t)$, which maps the state s_t to the action $a_t = \{\omega_t, \beta_t\}$. The architecture of the actor network consists of the following layers:

- **Input Layer**: The input to the actor network is the state vector $s_t = \{I_t, I_{t,i}, \omega_{t-1}, h_t\}$, which includes macroeconomic variables, firm-specific characteristics, previous portfolio weights, and the hidden state h_t from the Long Short-Term Memory (LSTM) network.
- **Hidden Layers**: The hidden layers are composed of fully connected layers with non-linear activation functions. We use the *Rectified Linear Unit (ReLU)* activation function:

  $$\text{ReLU}(x) = \max(0, x)$$

  The hidden layers allow the network to capture non-linear relationships and interaction effects between macroeconomic variables and asset characteristics.
- **Output Layer**: The output of the actor network is the action $a_t = \{\omega_t, \beta_t\}$, which consists of the portfolio weights and risk loadings. The action space is continuous, and we use a *softmax* function to ensure that the portfolio weights sum to 1:

  $$\omega_t = \text{softmax}(W^\top s_t + b)$$

  where W and b are the weight matrix and bias term for the output layer.

6.1.2 Critic Network

The *critic network* estimates the value function $V^\pi(s_t)$, which represents the expected cumulative reward for a given state s_t under policy π. The architecture of the critic network is similar to the actor network:

- **Input Layer**: The input is the same state vector $s_t = \{I_t, I_{t,i}, \omega_{t-1}, h_t\}$ used in the actor network.
- **Hidden Layers**: Fully connected layers with ReLU activation are used to model non-linear relationships between state variables and the value function.
- **Output Layer**: The output is the scalar value $V^\pi(s_t)$, representing the expected cumulative reward from state s_t. No activation function is applied to the output layer.

6.2 Training Procedure

The RL model is trained using a gradient-based optimization method to adjust the parameters of the actor and critic networks. We adopt the *Proximal Policy Optimization (PPO)* algorithm, a policy gradient method known for its stability and efficiency in training RL models. The training process consists of the following steps:

1. **Collect Experience**: The agent interacts with the environment (the financial market) by observing the state s_t, selecting an action a_t, and receiving a reward R_t. The experience tuple (s_t, a_t, R_t, s_{t+1}) is stored in a replay buffer.
2. **Compute Advantage**: The advantage function $A(s_t, a_t)$ is computed as the difference between the actual reward and the expected value from the critic network:
$$A(s_t, a_t) = R_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$
where $\gamma \in (0, 1]$ is the discount factor that controls the weight given to future rewards.
3. **Policy Update (Actor Network)**: The policy is updated by minimizing the loss function:
$$L_{\text{actor}}(\theta) = -\mathbb{E}\left[A(s_t, a_t) \log \pi_\theta(a_t \mid s_t)\right]$$
where θ are the parameters of the actor network.
4. **Value Function Update (Critic Network)**: The critic network is updated by minimizing the mean squared error (MSE) between the predicted and actual value function:
$$L_{\text{critic}}(\phi) = \mathbb{E}\left[\left(R_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)\right)^2\right]$$
where ϕ are the parameters of the critic network.
5. **Repeat**: The process is repeated over multiple episodes until the policy converges to an optimal strategy.

6.3 Regularization Techniques

To prevent overfitting, we apply several regularization techniques during training:

- **Dropout**: Dropout is used in the hidden layers of both the actor and critic networks to prevent overfitting. During training, a fraction of the neurons in each hidden layer is randomly set to zero, which helps the network generalize better.
- **Early Stopping**: We monitor the performance of the model on a validation set during training. If the validation performance stops improving after a certain number of epochs, training is stopped to prevent overfitting.
- L_2 **Regularization**: A penalty is added to the loss function based on the L_2 norm of the weights. This encourages the model to learn simpler, more generalizable representations.

6.4 Hyperparameter Tuning

Hyperparameters such as the learning rate, discount factor γ, and the number of hidden layers and neurons in each layer are tuned using cross-validation. The following hyperparameters are optimized:

- **Learning Rate**: The step size used by the gradient descent algorithm to update the network weights.
- **Discount Factor (γ)**: Controls the weight given to future rewards. A higher γ encourages the model to prioritize long-term rewards.
- **Batch Size**: The number of experience tuples used in each training iteration.
- **Number of Hidden Layers and Neurons**: The depth and width of the actor and critic networks are tuned to capture the complexity of the asset pricing problem.

6.5 GAN Integration

The Generative Adversarial Network (GAN) from *Chen, Pelger, and Zhu (2023)* is integrated into the RL framework to enhance the model's ability to minimize pricing errors. The GAN consists of two networks:

- **Generator**: The generator network in the GAN generates synthetic data points, which represent the hardest-to-price moments. These moments are selected to maximize pricing errors.
- **Discriminator**: The discriminator network evaluates the pricing performance of the agent's policy by comparing real and synthetic pricing errors.

At each time step, the GAN adversary generates a synthetic moment condition designed to maximize pricing discrepancies. The RL agent then adjusts the portfolio weights ω_t and risk exposures β_t to minimize the pricing error in response to these synthetic challenges. This adversarial setup ensures that the RL model learns to handle the most difficult pricing conditions, thereby improving its robustness.