Financial Training for Deep Neural Networks

Author: Babacan Guven

Supervisor: Nikolay Nikolaev
Goldsmiths, University of London
Department of Computing
BSc Computer Science

Abstract

High profits come with high risks, but what if we were able to obtain high profits with minimal risk? This project investigates the use of Neural Networks (NNs) to optimise the Sharpe Ratio (SR), a key measure of risk adjusted performance. Aiming to achieve high returns with reduced risk, the NNs output asset allocation values, where, for example, 0.1 indicates investing 10% of the portfolio, and -0.1 suggests selling 10%. The NNs aim to beat the S&P500 (^GSPC) benchmark while maintaining a good SR value. The NNs are trained and tested on the S&P500, with a configuration of specific features, using historical financial data, with performance validated through back testing and statistical evaluation metrics. This project is unique in that it optimises a financial performance metric directly through backpropagation, rather than relying on traditional loss function like the Mean Squared Error (MSE). If successful, this method could offer a promising direction for risk-aware algorithmic trading and portfolio optimisation.
Keywords: Neural Networks, Backpropagation, Times Series, Sharpe Ratio

I. Introduction

Risk Management is immensely predominant in the field of finance, particularly as financial markets grow more complex and data driven. With the emergence of Machine Learning (ML) techniques being applied, especially NNs, unique paths have been unveiled in modelling and decision-making. This paper explores the application of neural architectures – ranging from Single-Layer-Perceptrons (SLPs) to Multi-Layer-Perceptrons (MLPs) to more complex Deep Learning Models (DLMs). By leveraging backpropagation techniques, these models optimise the SR, a key performance measure in finance – this is what makes it interesting, as it doesn't rely on traditional metrics like Mean Squared Error (MSE). Furthermore, the framework naturally extends to the realm of Reinforcement Learning (RL), where the environment encompasses the market features (state), portfolio allocation (actions), and financial returns (rewards).

Motivation

Financial markets are intrinsically noisy, complex, unpredictable, and can be influenced by many external factors. Traditional statistical methods and theories struggle to deal with non-linear relationships within financial time-series. Because of this, there has been a shift to more adaptive methods such as ML, which can model complex patterns and generalise to unseen market conditions.

The intersection of technology and quantitative finance presents difficult problems, promoting exciting research and experiments, motivating this paper into NN based portfolio strategies.

Sharpe Ratio

The SR is a statistical metric that measures the risk adjusted return, which can be used to evaluate models and trading strategies - it quantifies the return of an investment relative to its risk. By optimising the SR, we maximise returns and minimise risk. It is calculated by subtracting the risk-free rate of return from the return of the portfolio, and dividing this by the standard deviation of the portfolio returns. Below is the formula:

$$SR = \frac{R_p - R_f}{\sigma_p}$$

where:
R_p = portfolio return
R_f = risk-free rate
σ_p = standard deviation of portfolio returns
[1]

Neural Networks

NNs are very well suited to difficult financial tasks, as they have the capacity to approximate non-linear functions. With their ability they can uncover intricate patterns within the financial data, that could not be achievable through traditional methods. In addition, their adaptability and scalability make them even more useful. *[2]*

Objectives

This paper aims to compare the effectiveness of different neural architectures in optimising portfolio strategies based on the SR, and to demonstrate the impact of ML techniques within the context of finance.

- Develop and implement various NN architectures for asset allocation and trading
- Optimise the SR through backpropagation
- Implement basic buy and sell strategies based on model predictions
- Assess whether the model can outperform the S&P500 benchmark
- Achieve a good SR score for performance evaluation

II. Background

Neural Networks

A NN is inspired by the structure and function of the human brain. Much like how biological neurons receive, process, and transmit information through synapses, artificial neurons receive inputs, apply transformations, and propagate signals to subsequent layers. These networks can adapt to their environments by adjusting internal parameters making them powerful tools for learning from data. NNs are particularly useful in uncovering intricate, nonlinear relationships that traditional models or formulas may fail to capture. Their ability to generalise from patterns in data is what makes them valuable in complex domains like financial forecasting and asset allocation. A key component enabling this capability is the activation function, which introduces nonlinearity into the network. In this work, the hyperbolic tangent *(tanh)* activation function is used, which outputs values between -1 and 1. This bounded range is well-suited for financial models where asset allocation decisions need to reflect both long (positive weight) and short (negative weight) positions. *[2]*

Single-Layer Perceptron (SLP)

The SLP is the simplest form of a neural network, consisting of a single neuron connected directly to the input. Normally used for linearly separable classification tasks, it has limited capacity to capture complex patterns. However, when combined with an activation function such as *tanh*, it can serve a role in financial settings by outputting values for asset allocation. In this project, the SLP demonstrates how even minimal architectures can learn basic relationships, particularly when smoothed prices are used as inputs, which appear to simplify the learning process. *[2]*

Multi-Layer Perceptron (MLP)

The MLP introduces one or more hidden layers, enabling the model to learn nonlinear patterns. Each layer extracts complex patterns, which is particularly valuable in financial environments that exhibit noise, volatility, and non-stationarity. In this context, the MLP has the capacity to identify meaningful trends and correlations that may inform profitable asset allocation strategies. Its structure strikes a balance between complexity and generalisation, often outperforming simpler models while maintaining robustness. *[2]*

Deep Learning Model (DLM)

The DLM extends the MLP with a greater number of hidden layers, allowing for deep feature extraction. In theory, the DLM should be capable of learning even more nuanced or hierarchical patterns within market data. However, with increased depth comes the risk of overfitting, particularly when the dataset is small or not sufficiently diverse. This is why the model used here is pruned - removing neurons based on probability - to reduce complexity and encourage other neurons to find patterns, and not just fit noise. Furthermore, although these networks do not explicitly predict asset prices, their allocation outputs implicitly reflect price direction forecasts. A higher allocation implies an expectation of upward price movement, and vice versa, linking neural decision-making to directional market views.

Gradient ascent and backpropagation

Gradient ascent is the optimisation method used in this project, as the objective is to maximise a financial metric - in this case, the SR. Unlike the more common gradient descent, which minimises a loss function, gradient ascent updates the model's parameters in the direction that increases the chosen objective.The gradients are computed using backpropagation, a fundamental algorithm in NNs that efficiently calculates the partial derivatives of the output with respect to each weight by applying the chain rule through the layers. This allows the network to adjust weights in a structured manner, reinforcing connections that contribute positively to the SR and diminishing those that do not. Together, gradient ascent and backpropagation form the learning engine that enables the model to iteratively improve its financial performance. *[2]*

Reinforcement Learning for Trading

According to the predominant work by *Moody* and *Safell* on Reinforcement Learning for Trading, RL is seen as a promising approach for optimising trading strategies by directly targeting performance metrics like risk-adjusted returns. RL can be assured as a promising approach because the results, in the paper, for reinforcement learning trading systems outperform the S&P 500 Stock Index over a 25-year test period. Continuing with *Moody* and *Safell's* work, two different reinforcement learning methods were applied. The first, Recurrent Reinforcement Learning, which uses immediate rewards to train the trading systems, while the second Q-Learning approximates discounted future rewards. *[3]*

The paper *"Reinforcement Learning for Systematic FX Trading"* by Borrageiro, Firoozye, and Barucca presents a Q-learning-based framework for developing automated trading strategies in the foreign exchange market, with a strong focus on maximising risk-adjusted returns through a carefully designed reward function that incorporates the SR. *[4]*

This directly relates to my own work, where I developed an asset allocation model using NNs that outputs portfolio weights in the range [-1, 1], effectively signalling short, neutral, or long positions across different assets. Like in their approach, I use the SR as a feedback mechanism to guide the training process and assess model performance. While their method relies on tabular Q-learning and discrete actions in FX trading, my model leverages continuous outputs and is designed to generalise across asset classes using deep learning. Both works share a common emphasis on building systematic trading strategies that adapt to market dynamics and optimize for long-term, risk-adjusted profitability.

RL can adapt to different levels of risk and volatility, which is vital when maximising the SR. Using a reward function based on the SR encourages the model to prioritise returns

with minimal risk, rather than maximising return alone which can come with a lot of risk.
There is a risk of overfitting when using RL for trading as financial markets are noisy, and RL models trained over specific historical periods may not generalise well to future market conditions, especially when there is extreme volatility. In addition, financial markets are non-stationary, meaning that the optimal trading strategy can change over time - techniques like continuous learning or online learning could enable the RL model to adapt to new data without forgetting past information.

Sharpe Ratio Maximisation

The SR is a simple but effective financial metric that calculates risk-adjusted performances of algorithms and helps assess the performance of an investment relative to its risk. SR is calculated by subtracting the risk-free rate of return from the return of the portfolio, and dividing this by the standard deviation of the portfolio returns. The output of the SR is considered as so:

- - 1 or above: acceptable or good
- - 2 or above: very good
- - 3 or above: excellent

Maximising the SR will be very beneficial because it can suggest better returns for each unit of risk taken, find the optimal balance of assets in a portfolio and reflect favourability of trading algorithms.

In the context of RL, a model that maximises the SR focuses on maximising profits with minimal risk. A notable example using the SR as a primary metric is presented in the work by *Choey and Weigend 1997*, where the trading strategy is based on the SR and not minimising the sum-squared error (SSE), which is more commonly used and an alternative to evaluate a trading strategy. In this research the SR is the objective function - assuming there are two asset classes, risky and risk-free, and on a specific period, the optimal allocation for the risky asset is selected to maximise the SR. The allocation inherits continuous values between [-1, 1]: a positive allocation signals a long position, meaning the asset's value is expected to increase over time, while a negative allocation indicates a short position, suggesting the asset is anticipated to decline in price. In a short position, the asset is borrowed and sold with the intention to repurchase it later at a lower price, profiting from the price difference. *[5]*

While focusing on the SR is beneficial for risk-adjusted performance, there are limitations to relying solely on this metric. The SR may not fully capture tail risks or sudden shifts in market volatility, potentially leading to unexpected losses during extreme market events. Additionally, focusing on risk-adjusted returns could reduce the model's focus on maximising absolute returns, which might be necessary in certain market conditions.

S&P500

Standard and Poor's 500 is a stock market index tracking the stock performance of 500 leading companies listed on stock exchanges in the United States. It is often used as a benchmark for the overall health of the U.S. equity market due to its broad coverage and historical reliability. The index generally exhibits a long-term upward trend. Using the S&P 500 for experimentation offers several advantages: its relative stability ensures fewer extreme fluctuations, which aids in training more robust and generalisable models. Additionally, comparing the model's performance against the S&P 500 benchmark allows for a meaningful assessment of strategy effectiveness in a real-world financial context. *[1]*

Deep Learning for Financial Applications

In the survey "Deep Learning for Financial Applications", the authors highlight how deep learning is increasingly used in algorithmic trading and portfolio management to extract complex patterns and optimise financial decisions. Techniques such as LSTMs and CNNs are applied to historical market data for signal generation, while deep reinforcement learning (DRL) models learn trading strategies that maximise long-term rewards. In portfolio management, DRL and NNs are used to predict asset returns and allocate weights dynamically based on risk-adjusted metrics like the SR. *[6]*

Reflecting this trend, the DLM in this paper outputs allocation weights between -1 and 1 for each asset, representing short or long positions. It evaluates performance using the SR as a reward signal and continuously retrains to improve portfolio efficiency. This aligns with the paper's observation that DL models can dynamically adjust to market conditions and optimise portfolios based on non-linear dependencies, despite challenges such as data noise and model interpretability.

Financial Training Criterion

In his paper *"Using a Financial Training Criterion Rather than a Prediction Criterion,"* Yoshua Bengio argues that optimising financial models directly for performance metrics - such as net returns or risk-adjusted gains - is more effective than training solely to minimise prediction error. This perspective resonates strongly with my own work, where I developed NNs that are asset allocation models trained to maximise the SR, rather than to predict future prices. By treating the SR as the objective function, my model learns to allocate portfolio weights within the range [-1, 1] -indicating short or long positions - in a way that directly improves financial outcomes. Like Bengio's approach, my methodology acknowledges that in financial markets, minimising predictive loss does not always translate to better trading performance. Instead, aligning the learning objective with real-world financial criteria allows the model to navigate noisy and non-stationary environments more effectively. *[7]*

III. Methods

Model implementation

The weights for all models are initially uniformly distributed between *-1 and 1,* this avoids overweighting specific neurons early in training, which can be useful when dealing with volatile financial data. In addition, it ensures random exploration in the early training phase and prevents the model form being biased toward a particular strategy too soon. The Learning Rate (LR - η) varies across models as different LRs optimise the SR depending on the model and inputs. Epochs are set to 100, which is plotted against the SR depending on when the SR is the highest the epoch will be limited to that number.

For all models, during training, many things are calculated such as: Allocation (output of the model - α), Asset Returns *(ξ_t),* Average Daily Return *($\bar{r}$),* Average Daily Excess Return *($\bar{r}_{excess}$),* Standard Deviation *(σ),* Annualised SR *(aSR)* and Quantity *(Ct).* The Excess Risk-Free Rate *(K)* is set to 0.005, and Training Days *(T)* is set to 253. *[5]*

- Asset Returns *(ξ_t)*: $\alpha_t \times r_t$
- Average Daily Return *($\bar{r}$)*: $\frac{1}{N}\sum_{t=1}^{N} \xi_t$
- Average Daily Excess Return *($\bar{r}_{excess}$)*: $\bar{r} - K$
- Standard Deviation *(σ)*: $\sqrt{\frac{1}{N}\sum_{t=1}^{N}(\xi_t - \bar{r})^2}$
- Annualised SR *(aSR)*: $\sqrt{T} \times \frac{\bar{r}}{\sigma}$
- Quantity *(Ct)*: $\frac{\sqrt{T}}{\sigma}(1 - (\frac{\bar{r}_{excess}}{\sigma} \times (\xi_t - \frac{\bar{r}}{\sigma})))$

Before getting onto the models themselves, the derivative of the SR with respect to the weights (SR update rule) is *[5]*:

$$\frac{\partial SR_T}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} Ct_i \times r_i \times \frac{\partial \alpha_i}{\partial w}$$

Gradient Ascent is performed, as we want to maximise the SR objective function, to update the models' parameters. The formula provided above is a global cost function, to update the parameters stochastically this rule is followed *[5]*:

$$\Delta w = Ct_i \times r_i(1 - \alpha_i^2)\gamma_i$$

where,

- γ: is the net-input to the final neuron in the model

It is important to note that only the weight to the final neuron is updated this way. All other neurons follow the normal backpropagation updates, using the derivative of the activation function *tanh*:

$$\delta_j = (1 - tanh^2(z)) \sum_k w \times \delta_k$$

$$\Delta w = w - \eta \cdot \delta_j \cdot a$$

where,

- w: weights from input to hidden
- a: input to corresponding hidden neuron

Implementation follows these steps:

1. Feed-forward pass through the network calculates the allocations for every day in the training data
2. These are calculated: ξ, $\bar{r}$, $\bar{r}_{excess}$, σ, *aSR, Ct*
3. A random pattern and the pattern with the highest return (best pattern) is selected – done separately
4. $\frac{\partial SR_T}{\partial w}$ is calculated
5. Parameters are updated
6. Repeat

For every loop in the epoch: allocations (output of the model) and the net-input (input to the final node – used in backpropagation) are arrays set to zeros of the length of the training data. Then, another loop goes through each input from the training data, and the dot product of the inputs and the weights is calculated, assigning the net inputs and allocations for each day.

Also, within training, for backpropagation, all models have two versions: random update and best update. The random update randomly picks an Asset Return to be used, and the best update uses the highest Asset Return to be used, the index of these returns will be taken as they will correspond accordingly to factors needed like the arrays of the net-inputs and allocations – the backpropagation will be explained thoroughly for each model below.

SLP:

Forward pass: In the forward pass of the SLP, each input vector is multiplied by a corresponding weight, and these weighted inputs are summed together. A single bias term is added to this total. The sum is then passed through a *tanh* activation function, which transforms the output into a value between -1 and 1. Because there are no hidden layers, all computation happens in a single step, making the SLP fast and easy to train. However, this simplicity also limits its capacity to learn complex, non-linear relationships, which is why it tends to perform better when the inputs are smoothed or otherwise simplified.

Backpropagation: After computing the output and evaluating the resulting financial returns over the training period, the SR is calculated. The key idea in backpropagation for the SLP is to determine how much each input weight contributed to the final output and, more importantly, how those contributions influenced the SR. In this context, backpropagation involves calculating the derivative of the SR with respect to each input weight. Since there are no intermediate layers, the gradient computation is relatively straightforward - we only need to assess how a small change in each weight alters the output allocation and how that in turn affects the SR. This direct relationship allows for a more transparent interpretation of how the model is learning: increasing a weight strengthens the influence of that input on the final

decision, while decreasing it reduces its impact. The tanh activation function plays an important role here. Its derivative - which depends on the output value - acts as a scaling factor, controlling how large the weight updates should be.

Weight update: Once the gradients are computed, the weights are updated using gradient ascent, which is used instead of the more common gradient descent because the goal is to maximise the SR, not minimise a loss. Each weight is adjusted slightly in the direction that improves the SR, using a predefined learning rate to control the step size. If a particular input is found to have a positive impact on risk-adjusted return, its associated weight is increased; if it's detrimental, the weight is reduced.

MLP:
Forward pass: During the forward pass, the MLP processes each input and passes it through the network to produce an output. The process begins with the input layer feeding values into the hidden layer, where a set of weights and biases transform the input through the hyperbolic tangent. This activation ensures that outputs remain bounded between -1 and 1. The output from the hidden layer is then passed into a final neuron, which again uses the *tanh* activation function to produce the allocation value.

Backpropagation: the derivative of the SR is computed with respect to the final output neuron's weights. This step is crucial because it identifies how sensitive the SR is to changes in the output allocation. The model also needs to adjust weights in the hidden layer. To do this, backpropagation uses the chain rule to trace the contribution of each hidden layer weight to the final output and, in turn, to the SR. These hidden layer updates help the network refine internal feature representations so that over time, it learns which combinations of financial signals are most predictive of risk-adjusted returns.

Weight update: with the gradients calculated, the weights are then updated using gradient ascent. This process means the weights are nudged in the direction that increases the SR, rather than decreases a loss. The learning rate controls how big each adjustment is. This approach allows the MLP to iteratively improve its allocation decisions. Over many epochs, the network adapts its internal parameters to prioritise inputs and patterns that consistently lead to higher SRs.

DLM:
Forward pass: just like in the MLP, the forward pass in the DLM begins with the input features, which are passed through the first hidden layer. The output of this layer is then passed into a second hidden layer before reaching the final output neuron. Each transition between layers involves weight matrices and biases that are transformed by the tanh activation function. The addition of the second hidden layer allows the DLM to capture more intricate dependencies and interactions that the MLP might miss.

Backpropagation: in the backward pass, the training logic is extended to accommodate the additional hidden layer. First, the sensitivity of the SR to changes in the final output is computed. Then, the model computes how the weights in the second hidden layer contributed to that output. This involves propagating the error signal one layer deeper, calculating the influence of the second layer's outputs on the final result. After that, the influence of the first hidden layer is evaluated. This cascading of error signals backward through the network - from output to second hidden, and then from second hidden to first hidden - enables each set of weights to be updated in a way that considers not only their direct effect on the output but also their indirect contribution through subsequent layers. This step is more computationally intensive than in the MLP, and it is also more sensitive to issues like vanishing gradients, especially when using activation functions like *tanh*. However, because this model is trained with relatively shallow depth (only two hidden layers), those concerns are mitigated, especially with careful weight initialisation and a controlled learning rate.

Weight update:
As in the MLP, the computed gradients are used to adjust the weights through gradient ascent. Because of the deeper structure, more sets of weights are involved - one for each layer. Each update is designed to push the network toward configurations that increase the SR. The addition of the second hidden layer makes the DLM potentially more powerful, but it also increases the risk of overfitting. To address this, the project incorporates pruning strategies - assigning each neuron a probability of being dropped out during training - which helps reduce model complexity and improve generalisation.

SLP:

Inputs: $X \in \mathbb{R}^{1\,x\,d}$

Weights: $W \in \mathbb{R}^{d\,x\,1}$

Bias: $b \in \mathbb{R}^{1\,x\,1}$

Feedforward:

$Z = X \cdot W + b$

$\alpha_t = \tanh(Z)$

Backpropagation:

$\beta_{Out} = Ct \times Data[pattern] \times (1 - tanh^2(\alpha[pattern]) \times Z[pattern])$

$\Delta w = \beta_{Out} \times \eta$

$\Delta b = \beta_{Out} \times \eta$

$w = w + \Delta w$

$b = b + \Delta b$

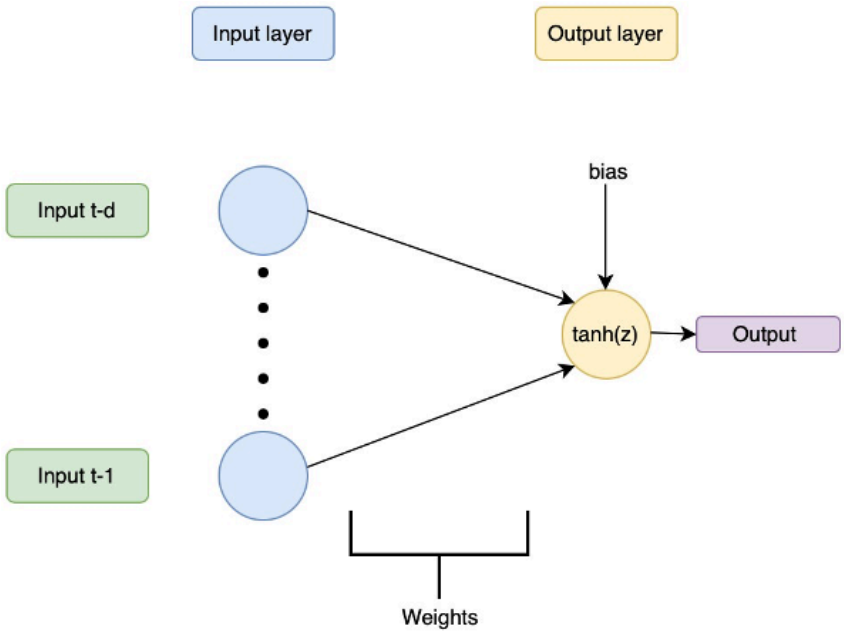

Figure 1 - SLP architecture

MLP:

Inputs: $X \in \mathbb{R}^{1\,x\,d}$

Weights (Input to Hidden - w_{ij}): $W^1 \in \mathbb{R}^{d\,x\,h}$

Weights (Hidden to Output - w_{jk}): $W^L \in \mathbb{R}^{h\,x\,o}$

Bias (Hidden): $b^1 \in \mathbb{R}^h$

Bias (Output): $b^L \in \mathbb{R}^o$

Feedforward:

$Y = \tanh(\mathrm{X} \cdot W^1 + b^1)$

$Z = Y \cdot W^L + b^L$

$\alpha_t = \tanh(Z)$

Backpropagation:

$\beta_{Out} = Ct \times Data[pattern] \times (1 - tanh^2(\alpha[pattern]) \times Z[pattern])$

$\beta_{Hidden} = (1 - tanh^2(Y)) \times (\mathrm{W^L} \times \beta_{\mathrm{Out}})^T$

$\Delta w_{jk} = \beta_{Out} \times \eta$

$\Delta w_{ij} = (X^T \cdot \beta_{Hidden}) \times \eta$

$\Delta b^L = \beta_{Hidden} \times \eta$

$\Delta b^1 = \beta_{Out} \times \eta$

$w_{jk} = w_{jk} + \Delta w_{jk}$

$w_{ij} = w_{ij} + \Delta w_{ij}$

$b^L = b^L + \Delta b^L$

$b^1 = b^1 + \Delta b^1$

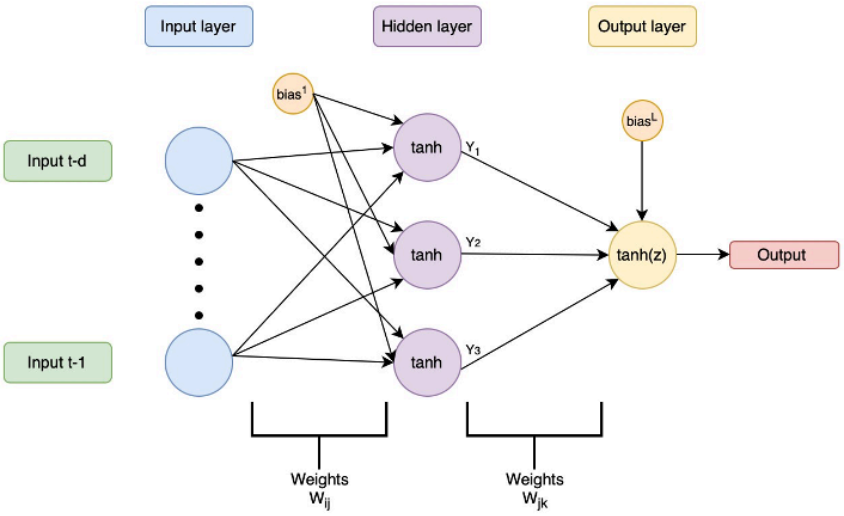

Figure 2 - MLP architecture

DLM:

Inputs: $X \in \mathbb{R}^{1\,x\,d}$

Weights (Input to 1st Hidden - w_{ij}): $W^1 \in \mathbb{R}^{d\,x\,h}$

Weights (1st Hidden to 2nd Hidden - w_{jk}): $W^2 \in \mathbb{R}^{h\,x\,s}$

Weights (2nd Hidden to output - w_{kl}): $W^L \in \mathbb{R}^{s\,x\,o}$

Bias (Hidden): $b^1 \in \mathbb{R}^h$

Bias (Hidden): $b^2 \in \mathbb{R}^s$

Bias (Output): $b^L \in \mathbb{R}^o$

Feedforward:

$V = \tanh(\mathrm{X} \cdot W^1 + b^1)$

$Y = \tanh(\mathrm{V} \cdot W^2 + b^2)$

$Z = Y \cdot W^L + b^L$

$\alpha_t = \tanh(Z)$

Backpropagation:

$\beta_{Out} = Ct \times Data[pattern] \times (1 - tanh^2(\alpha[pattern]) \times Z[pattern])$

$\beta_{Hidden2} = (1 - tanh^2(Y)) \times (\mathrm{W^L} \times \beta_{\mathrm{Out}})^T$

$\beta_{Hidden1} = (1 - tanh^2(V)) \times (\beta_{Hidden2} \cdot (W^2)^T)$

$\Delta w_{kl} = \beta_{Out} \times \eta$

$\Delta w_{jk} = (V^T \cdot \beta_{Hidden2}) \times \eta$

$\Delta w_{ij} = (X^T \cdot \beta_{Hidden1}) \times \eta$

$\Delta b^L = \beta_{Out} \times \eta$

$\Delta b^2 = \beta_{Hidden2} \times \eta$

$\Delta b^1 = \beta_{Hidden1} \times \eta$

$w_{kL} = w_{kl} + \Delta w_{kl}$

$w_{jk} = w_{jk} + \Delta w_{jk}$

$w_{ij} = w_{ij} + \Delta w_{ij}$

$b^L = b^L + \Delta b^L$

$b^2 = b^2 + \Delta b^2$

$b^1 = b^1 + \Delta b^1$

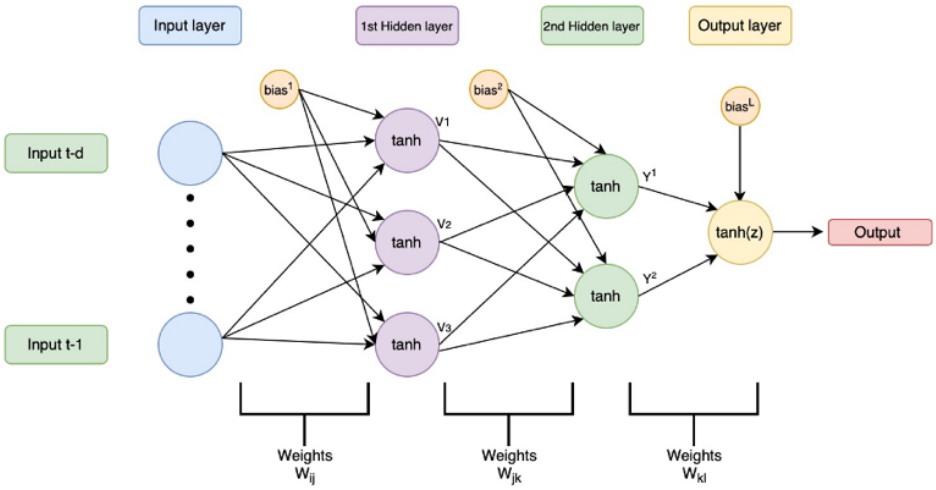

Figure 3 - DLM architecture

Testing

Data preprocessing
First, the financial data was deliberately left unscaled as preliminary experiments connoted that applying scaling techniques such as Min-Max Normalisation *[8]* and Standardisation *[9]* degraded model performance. This suggests that preserving the original form of the financial time series is critical in developing financial models, as it retains fundamental market dynamics, and that scaling can distort this information. In addition, everything was calculated using '*Close Prices'* – the final price of the asset of that day.

For the NNs, *tanh* activation function *[10]* was used because it outputs values in the range [-1,1], which is applicable for the aim of this paper, where the models output an allocation size. A positive output indicates a buy signal, and a negative output indicates a sell signal, where 0.4 would mean use 40% of portfolio to buy and -0.4 to sell 40% of portfolio, respectively. Additionally, *tanh* provides non-linearity, allowing the NN to learn more complex patterns in the input data.

The libraries used for data manipulation and visualisation were *Pandas, NumPy* and *Matplotlib. [11][12][13]*

Financial data was used from *'yfinance'* – Yahoo Finance *[14]* is a reliable source to get accurate information about assets, it gives information like *Open Prices, Close Prices and Volume.* The NNs were trained on the S&P500 data from *2018/01/01 – 2022/12/31.*

The extracted features included:
- PCT: Percentage returns (used for auto-regression)
- V: Volatility
- SMA: Simple Moving Average
- EMA: Exponential Moving Average

Several feature configurations were explored, including:
- PCT (Auto-regression only)
- PCT + V + SMA
- PCT + EMA

To store all the features for the model, a *Data Frame* would be created, using *Pandas*, with the necessary information to extract the relevant information.

PCT:
The returns were calculated using the Pandas function *'pct_change'*. This method calculates the return for the current day:

$$pct\_change\ (r_t) = \frac{\mathrm{X}_t - \mathrm{X}_{t-1}}{X_{t-1}}$$

where,
- X_t: current value
- X_{t-1}: previous value

As it calculates for the current day, this wouldn't be compatible for a financial model because we wouldn't know what the return would be for that day until the day is over, meaning no trades would be done. To resolve this, we need to shift the data, as we're looking at daily information its appropriate to shift the data by the amount of days you want to look at – in the Auto Regression only, we look at 7 days, so creating 7 columns each representing the past days you *shift* the data by that much to get the return for that day (it is not cumulative).

V:
Volatility, σ, represents how much the price of an asset is fluctuating, making it a very valuable feature. Higher volatility indicates larger fluctuations and greater uncertainty. Typically calculates as the standard deviation of returns:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{t=1}^{N}(r_t - \bar{r})^2}$$

where,
- r_t: return at time *t*
- $\bar{r}$: average return over the period
- N: number of days

After calculating PCT, volatility is easy to extract – using Pandas' *rolling* function, you declare the *window* representing the number of days, and chain this with NumPy's *std* function, which calculates the standard deviation – resulting in the volatility. Again, you will have to shift the data here, as calculations include the current day, which is not reflective of the real-world.

SMA:
The SMA calculates the average price of an asset over a period of days. It's a useful indicator as it identifies trends by reducing noise – for example if prices are above the SMA it may indicate an uptrend. In this case, we apply it to the returns, because it allows a better understanding of the average return trends over the selected period of days. Its calculation is simple:

$$SMA_t^{(n)} = \frac{1}{n}\sum_{i=0}^{n-1} r_{t-i}$$

where,
- $SMA_t^{(n)}$: SMA of returns at time *t* over a window of *n*
- r_{t-i}: return *i* days before time *t*
- n: number of days

Using the PCT, the *rolling* function is used again with the window initialised with the number of days you want, now chained with NumPy's *mean* function. In this paper, the SMA is calculated only for the most recent return. In addition, many windows were used here, which can be

implemented through a loop. Again, data was shifted here, only by 1.

EMA

The EMA is a weighted moving average that gives more importance to recent data points, making it more responsive to recent price or return changes compared to the SMA. EMA is extremely beneficial, because it can detect trends earlier. Its calculation:

$$EMA_t = \alpha \times r_t + (1 - \alpha) \times EMA_{t-1}$$

$$\alpha = \frac{2}{span + 1}$$

where,

- α: smoothing factor
- r_t: return at time *t*
- span: number of days

Smoothing:

For all configurations of all models, a smoothing factor was also applied to compare the results with models that used raw prices, to see the impact a smoothing factor can have. The smoothing formula used is a modified exponential smoothing formula that incorporates momentum into the smoothing process, while still filtering out excessive noise – this can help anticipate price acceleration and react faster to trends. The formula:

$$P_t = \alpha(X_t + 0.5(X_t - X_{t-2})) + (1 - \alpha)P_{t-1}$$

where,

- X_t: raw price at time t
- P_t: smoothed price at time t
- α: smoothing factor

Creating an array to store the *smoothed prices,* with the first raw price appended to it, with a for loop you can loop through the raw prices and calculate the smoothed price and append it to the array.

The testing phase serves as a critical component of this project, providing a realistic assessment of how well the trained NNs perform in a simulated, out-of-sample trading environment. While the models are optimised during training to maximise the SR, their actual performance can only be validated by applying them to unseen data and observing their decision-making over time. To this end, a simple but effective daily trading simulation was implemented using S&P500 data from the period 2023/01/01 to 2024/12/31.

The trading logic is deliberately kept straightforward to ensure that any gains or losses can be primarily attributed to the model's allocation decisions rather than to engineered strategy layers. The simulation begins with a portfolio of 100 shares and $100,000 in cash, and the model operates with 20% of the portfolio value as trading capital. This cap is introduced to manage risk, as deploying 100% of the capital could exaggerate gains or losses and potentially create a misleading evaluation of model behaviour. By limiting trading capital, the simulation more accurately mimics conservative trading practices that seek to preserve a capital buffer for unforeseen volatility or liquidity constraints.

Buy decisions are triggered when the model's output allocation is greater than zero, indicating a bullish sentiment. In such cases, shares are purchased using the available trading capital, ensuring that the amount spent does not exceed the cash reserves or the cost of available shares. Conversely, sell decisions are activated when the model's output is negative, signalling a bearish outlook. Here, the simulation ensures that the number of shares sold does not exceed the current holdings, avoiding any unrealistic or short-selling behaviours that could distort results.

In addition to simulating buy and sell actions, the testing phase meticulously tracks a range of performance metrics. These include daily profit and loss (P&L), cumulative returns, the total portfolio value (combining cash and the market value of held shares), and the daily model-generated allocations. This level of detail not only provides a clearer picture of how the model navigates the market but also supports meaningful comparisons between strategies across different architectures and feature sets.

Notably, all decisions are based on the same Close Prices used during training, maintaining consistency across all stages of the workflow. The purpose of this structure is to highlight the power of the neural network itself—without the influence of complex trading strategies, leverage, or external signals. By isolating the model's outputs, this phase tests the robustness and generalisability of the trained networks in responding to real market data.

Initial Setup:

- 100 shares and $100,000
- Portfolio Value = shares + cash
- Trading Capital = 20% of portfolio value (updated dynamically)
- Track Metrics: portfolio value, daily profit, cumulative returns and model allocations

Trading Logic:

- Buy (allocation > 0):
    - Use part of the trading capital to buy shares
    - Limited by available cash and share price
    - Multiple buy indication, or cash runs out, will result into a hold
- Sell (allocation < 0):
    - Sell shares proportionally to the negative allocation
    - Can't sell more than currently held

IX. Results

As three different configurations of inputs were used, with a smoothing factor being applied to compare with raw prices, and three models being created for each with two versions – this makes 36 models.

To provide greater transparency and facilitate comparison across all model configurations, a comprehensive results table has been included at the end of the paper. This table captures each model type (SLP, MLP, DLM), update method (Random, Best), input configuration (Auto Regression, PCT+V+SMA, PCT+EMA), and whether smoothing was applied. It lists both the final SR and total profit for each configuration. These results show consistent patterns supporting earlier conclusions; the MLP typically outperforms both SLP and DLM, with DLM occasionally matching or surpassing it under feature-rich, unsmoothed setups. Moreover, visual aids such as SR progression plots over epochs, can also be found at the end of the paper, offer intuitive views of model learning behaviour. In addition, the plots comparing the models' performance to the S&P500 benchmark can also be found at the end of the paper.

Only auto regression (PCT)
Without the smoothing factor being applied we can see the MLP with the random update produces the best SR with 1.66. The DLM with the random update produces the highest profits with a SR of 1.63. The perceptron with the random update performs poorly compared to the MLP and DLM. When the max return pattern update is applied, performance across all models becomes more uniform, with relatively small differences in both profits and SRs.

With the smoothing factor applied we can see huge improvements in the SLP, achieving a SR of 1.62 with the best update. The MLP performs the best in this case, with a SR of 1.65 with the best update and the highest profits. The DLM does quite poorly compared to the SLP and MLP, with a SR of 1.58 with the random update.

When using only auto-regressive inputs, the MLP consistently emerges as the most reliable model, especially when the smoothing factor is applied, offering strong and stable risk-adjusted returns.

PCT + V + SMA
Without the smoothing factor being applied it is clear to see that the best update consistently yields better SR than the random update for each model. DLM has the highest SR of 2.48, but the MLP with the highest profits has a SR of 2.47 – not much to distinguish between both. The SLP performs well with a SR of 2.41.

However, with the smoothing factor applied, we see a notable decline in performance in both profits and SR across all models. All models perform similarly with a SR between 2.30 to 2.34, but the MLP with the best update stands out with a SR of 2.37.

Overall, the combination of auto-regression, volatility, and SMA provides a strong foundation for predictive modelling, with all three architectures. Nonetheless, the MLP proves to be the most consistently profitable and reliable, excelling across both smoothed and unsmoothed conditions.

PCT + EMA
Without the smoothing factor applied, the poor performance of the SLP is apparent, achieving a SR of 1.37 as its best result (with best update). The best update produces the highest SRs across all models, with MLP achieving 1.70 and DLM achieving 1.68 – which is decent, and connotes improvements from using auto regression only.

When the smoothing factor is applied, SLP sees a modest yet meaningful improvement, increasing its performance to a SR of 1.48 across both versions. Performance decreases with the MLP and DLM, with SRs of 1.63 and 1.62 respectively.

In summary, adding EMA to auto-regressive inputs yields noticeable gains in performance, particularly for the MLP and DLM, and especially when no smoothing factor is applied. However, applying a smoothing factor appears to blunt the signal's responsiveness, mildly benefitting the SLP while reducing the edge that more complex models have. Overall, the MLP once again demonstrates the best balance between profitability and SR, with the DLM not far behind.

Summary
Across all configurations tested in this study - ranging from simple auto-regression to enriched inputs such as V, SMA and EMA several clear patterns emerge regarding model behaviour, input effectiveness, and the overall impact of smoothing price data.

The MLP consistently demonstrated the most robust and reliable performance across nearly all input configurations, particularly when enhanced with additional features. It often produced the highest profits and among the strongest SR, making it the most dependable model. The DLM, a more complex architecture, also performed strongly but was sometimes outperformed by the MLP - particularly when overfitting or noise may have been an issue. The SLP generally lagged in raw performance but showed noticeable improvements under certain conditions.

One of the most significant findings was the impact of applying a smoothing factor to input prices. This generally led to:

- Improved performance for SLP, particularly under auto regressive and EMA conditions.
- Stabilised performance across all models, reducing variance between random and best updates.
- Slightly reduced performance for MLP and DLM in some scenarios — likely due to the loss of subtle,

short-term fluctuations which these more complex models are able to learn and exploit.

Input Feature Impact:

- Auto-regression alone offered a baseline for model comparison. MLP and DLM already outperformed SLP in this scenario, showing their superiority in capturing temporal dependencies.

- Adding volatility and SMA features significantly boosted performance across all models. Notably, DLM achieved a SR of 2.48 and MLP closely followed with 2.47, indicating that these features enhanced model predictive power. SMA helped models understand mean-reversion tendencies, while volatility likely captured momentum or risk-sensitive patterns.

- With EMA features, performance varied more. While MLP and DLM still performed reasonably well, the SLP's relative gain with smoothing highlighted again how simplified inputs help simpler architectures. EMA likely introduced trend sensitivity that, when smoothed, was more digestible for the SLP.

Random vs Best Update Strategies

In most scenarios, best update strategies consistently outperformed random updates, especially when features were rich, and smoothing was applied. This reaffirms the importance of model training procedures and optimisation for extracting the full potential of each model.

Results Conclusion

This analysis reveals that model complexity should be aligned with feature richness and data preprocessing. The MLP, with its balance of depth and generalisation ability, excelled across the board and showed resilience to input variations. The DLM showed powerful results when supported with enriched features but was more sensitive to noise and data preparation. The SLP, while limited, benefitted the most from smoothed inputs and simpler features, suggesting its suitability in low-resource or low-complexity forecasting environments. Ultimately, the best performing configurations combined multi-feature inputs (e.g., autoregression + volatility + SMA) with appropriate data preprocessing (smoothing), and well-tuned update strategies — particularly benefiting models like MLP that can leverage richer data representations.

X. Discussion

The findings from this project demonstrate clear differences in model effectiveness, feature utility, and the role of data preprocessing in financial forecasting and asset allocation. These results are best understood in the context of both the project goals and the broader body of literature in machine learning-driven finance.

Interpretation Within This Project
The MLP consistently outperformed the other architectures - SLP and DLM - across nearly all feature configurations. This supports the idea that a balance of depth and generalisation, as found in MLPs, offers the most practical value in noisy, non-stationary financial environments. Particularly, when using enriched inputs such as volatility and moving averages, the MLP demonstrated robust profitability and high SRs, with values peaking around 2.47.

Interestingly, the application of a smoothing factor improved SLP performance substantially. This aligns with the project's hypothesis that simpler models benefit more from pre-processed, denoised input data. Meanwhile, the more complex MLP and DLM models saw slight degradation with smoothed inputs, suggesting that these models may rely on short-term volatility signals that smoothing removes. These results highlight the importance of tailoring data preprocessing strategies to model complexity.

Comparison to Broader Literature
These findings closely align with conclusions in the literature advocating for aligning model objectives with financial performance metrics. The use of the SR as an optimisation objective - also emphasised in works by *Bengio and Borrageiro et al. [4][5]* - was validated here as an effective way to produce risk-aware trading strategies. As seen in *Choey and Weigend (1997) [7],* using SR as a learning signal leads to strategies that prioritise consistent, low volatility returns over high but erratic profits. This project reinforces that insight by demonstrating how models trained with this objective can outperform baseline benchmarks like the S&P 500 in simulated scenarios.

From a model architecture perspective, the results are consistent with current applications of deep learning in financial modelling. MLPs, being more resilient to noise and better able to extract nonlinear relationships, mirror the findings of the *"Deep Learning for Financial Applications: A Survey",* where such architectures are praised for their versatility in dynamic market settings. DLMs, while theoretically more powerful, showed tendencies toward overfitting - underscoring the necessity of pruning and regularisation, especially in datasets with limited diversity or size.

Contributions and Novel Insights
This work contributes empirical support to the growing consensus that increasing feature richness significantly enhances model performance, provided the model complexity is sufficient to capitalise on it. While the idea of using technical indicators as input features is not new, this study quantifies their impact across architectures under both smoothed and unsmoothed data regimes, offering practical insight into when and how to use them.

Moreover, the experiment comparing random vs best update strategies underscores the importance of optimisation procedures. Best update methods consistently outperformed random updates, particularly in feature-rich environments, reinforcing the role of carefully tuned gradient ascent in financial training criteria.

XI. Limitations and Future Work

While the algorithm was successful in optimising the SR and demonstrated promising performance across various NN architectures and feature combinations, several limitations were observed, and there is valuable space for promising developments and further research.

Lack of Incremental Learning
A significant limitation is the static nature of the training process during the testing phase. Once the model is trained, it is not updated with new data as time progresses. In a real-world trading environment, markets are dynamic and constantly evolving. Incorporating online or incremental learning - where the model is retrained or fine-tuned daily or weekly for a few epochs using the latest market data, this could massively improve adaptability and allow the model to adjust to changing trends and volatility. This would likely enhance both profitability and robustness.

Simplistic Trading Strategy:
The current implementation relies on a basic buy and sell depending on the models' output. This simplistic trading logic lacks nuance and could be enhanced to obtain higher profits. Future work could involve implementing more sophisticated trading strategies, such as *[15]:*

- Long and Short strategies with position-specific thresholds and risk tolerances.
- Mean Reversion or Momentum-based strategies that respond to statistical anomalies or price trends.
- Custom rule-based strategies that combine technical indicators like RSI and Bollinger Bands with NNs predictions.

This would enable potentially more profitable simulation of trading decisions.

Pruning and Model Optimisation
The DLM in this project was pruned, reducing the number of active neurons and improving model efficiency. However, this optimisation technique was not applied to the MLP. Exploring pruning for the MLP could potentially enhance its performance by reducing overfitting and forcing the model to focus on more meaningful patterns. Furthermore, combining pruning with regularisation techniques could lead to even more generalisable models. *[16]*

Asset and Market Limitations:
The dataset used in this study was exclusively based on the S&P 500 index. While this provides a controlled and standardised testing environment, it limits the generalisability of the results. Future work should test the algorithm on different asset classes like cryptocurrencies, commodities and forex, and across multiple market conditions such as bull, bear and high volatility. This would reveal how well the models generalise and adapt to varying financial instruments. *[17]*

Model Architecture Expansion:
The study focused on SLPs, MLPs and DLMs. Future extensions could explore the application of more complex models such as:

- Recurrent Neural Networks (RNNs) *[18]* or Long Short-Term Memory (LSTM) *[19]* networks for capturing temporal dependencies in time-series data.
- Convolutional Neural Networks (CNNs) *[20]* for extracting local patterns in financial signals.
- Reinforcement Learning approaches such as Q-Learning *[21]* to enable an agent to learn trading policies through interaction with the market environment.

Feature Engineering and Enrichment:
While the study explored combinations of auto-regression, moving averages (SMA and EMA), and volatility, further work could investigate the impact of:

- Sentiment analysis from news or social media. *[22]*
- Macroeconomic indicators (interest rates, inflation data). *[23]*
- Alternative technical indicators or even raw limit order book data. *[24]*

Adding richer contextual features could allow the models to make more informed and nuanced predictions.

Evaluation Metrics and Risk Considerations:
Although the SR was a useful metric for this study, future research could incorporate additional performance measures such as:

- Maximum drawdown: to assess downside risk. *[25]*
- Calmar or Sortino Ratios: which are more sensitive to negative returns. *[26]*
- Transaction costs and slippage: to simulate realistic trading environments. *[27]*

These additions would provide a more comprehensive evaluation of model performance in real-world trading scenarios.

XII. Conclusion

This project has demonstrated that NNs can be trained directly to optimise financial performance metrics, such as the SR, yielding promising results in the realm of portfolio allocation and algorithmic trading. Through extensive experimentation with different architectures and feature configurations, it was shown that models can learn not only to interpret financial signals but also to make allocation decisions that reflect risk-adjusted performance goals.

One of the key takeaways is that model complexity must be appropriately matched with input feature richness and data preprocessing strategies. The MLP stood out as the most balanced and reliable model, often outperforming both the simpler SLP and the more complex DLM. Its ability to capture relevant patterns without overfitting suggests that moderate depth combined with careful training yields the best generalisation in dynamic financial environments.

The use of smoothing techniques proved especially beneficial for simpler architectures like the SLP, demonstrating that preprocessing choices must be tailored to model complexity. Meanwhile, models like the DLM, which are capable of learning nuanced temporal dependencies, benefitted more from raw data that preserved short-term signals. This balance between raw and processed input reflects a broader theme: success in financial ML depends not just on the model itself, but also on how well the data is prepared and aligned with the model's learning capacity.

Moreover, by using the SR as the objective function, this project was able to align model training with real-world financial goals. Rather than focusing on predictive accuracy or statistical fit, the models learned to produce returns that were both strong and stable, relative to the level of risk taken. This reframing of the training process aligns with current trends in financial ML research, where emphasis is increasingly placed on end-to-end financial outcomes rather than intermediate predictive metrics.

While the study involved certain limitations - such as the absence of real-time learning, limited market diversity, and basic trading logic - it lays a solid foundation for future work. The results point toward several promising directions, including the incorporation of adaptive or online learning, exploration of richer features, and deployment across broader asset classes. Additionally, expanding evaluation metrics to include drawdowns, transaction costs, and alternative risk measures could further improve the realism and applicability of such systems in professional trading environments.

In conclusion, this work provides empirical evidence that NNs, when trained with a clear financial objective like the SR can produce effective and interpretable portfolio allocation strategies. With further refinement, such models hold real potential for deployment in intelligent trading systems and risk-sensitive investment tools.

	Random pattern update		Max return pattern update	
S&P500	Profits	Sharpe Ratio	Profits	Sharpe Ratio
Perceptron	$196251.07	1.51	$206826.12	1.54
MLP	$229044.89	1.66	$206771.82	1.56
DLM	$238746.34	1.63	$213642.30	1.55

Table 1.1 - Results from PCT (auto-regression only)

	Random pattern update		Max return pattern update	
S&P500	Profits	Sharpe Ratio	Profits	Sharpe Ratio
Perceptron	$192649.41	2.35	$200990.39	2.41
MLP	$198965.92	2.38	$209769.88	2.47
DLM	$208728.41	2.43	$209763.19	2.48

Table 1.2 - Results from PCT+V+SMA

	Random pattern update		Max return pattern update	
S&P500	Profits	Sharpe Ratio	Profits	Sharpe Ratio
Perceptron	$128862.67	1.29	$153570.53	1.37
MLP	$225715.09	1.62	$235028.32	1.70
DLM	$238724.10	1.67	$228361.67	1.68

Table 1.3 - Results from PCT+EMA

	Random pattern update		Max return pattern update	
S&P500	Profits	Sharpe Ratio	Profits	Sharpe Ratio
Perceptron	$220034.08	1.59	$229343.31	1.62
MLP	$224408.48	1.64	$231524.42	1.65
DLM	$220826.85	1.58	$218144.76	1.56

Table 2.1 - Results from PCT - Smoothed prices

	Random pattern update		Max return pattern update	
S&P500	Profits	Sharpe Ratio	Profits	Sharpe Ratio
Perceptron	$193641.27	2.30	$199449.74	2.34
MLP	$195047.14	2.32	$203377.57	2.37
DLM	$196920.70	2.33	$198174.96	2.34

Table 2.2 - Results from PCT+V+SMA - Smoothed prices

	Random pattern update		Max return pattern update	
S&P500	Profits	Sharpe Ratio	Profits	Sharpe Ratio
Perceptron	$184131.53	1.48	$185245.08	1.48
MLP	$233133.67	1.63	$233800.99	1.63
DLM	$181147.83	1.55	$188712.99	1.62

Table 2.3 – Results from PCT+EMA - Smoothed prices

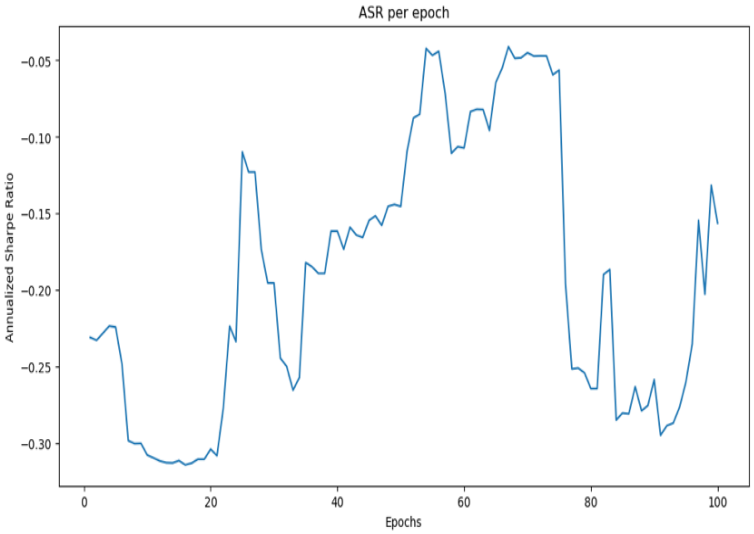

Figure 4 - SLP; PCT- Random update

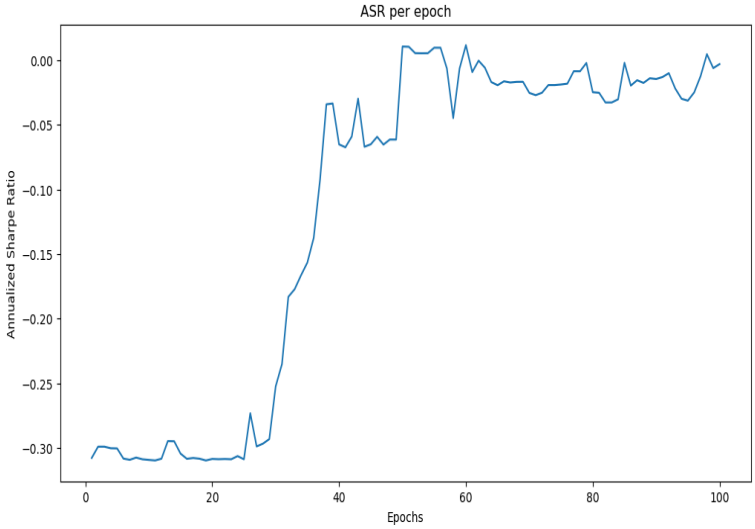

Figure 4.1 – SLP; PCT, EMA - Random update

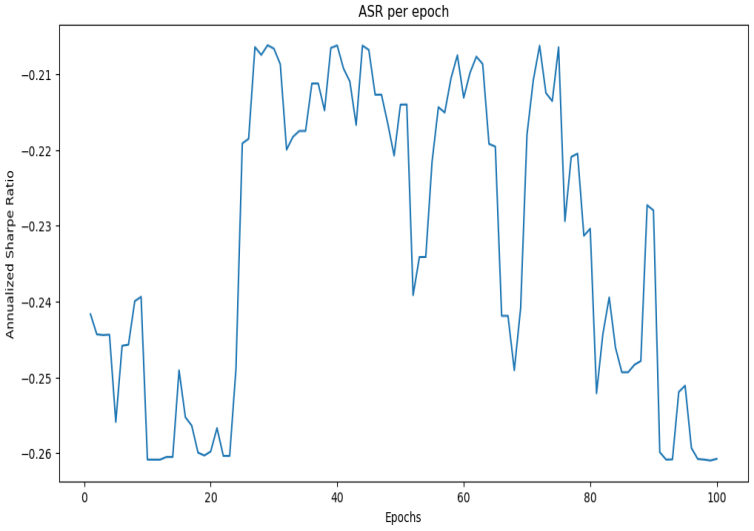

Figure 4.2 - SLP; PCT, V, SMA - Random Update

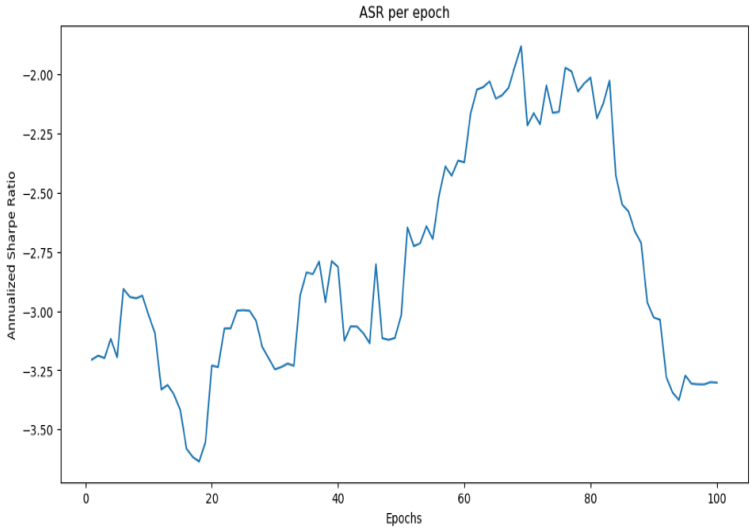

Figure 4.3 - SLP; PCT, Smoothed - Random update

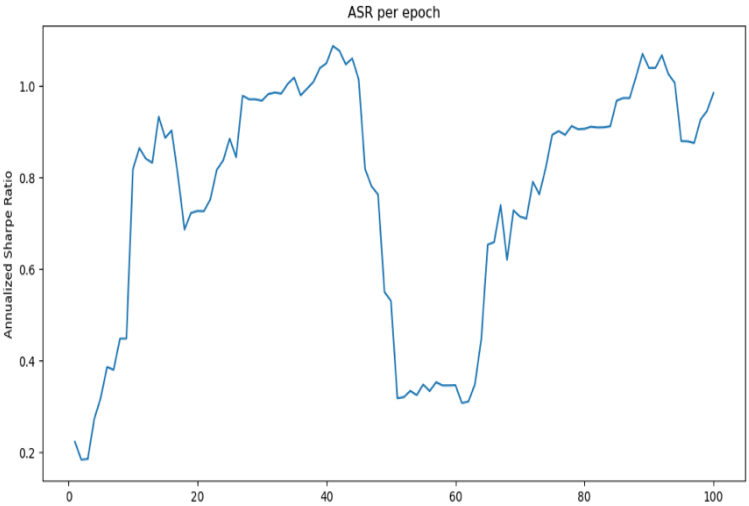

Figure 4.4 - SLP; PCT, EMA, Smoothed - Random update

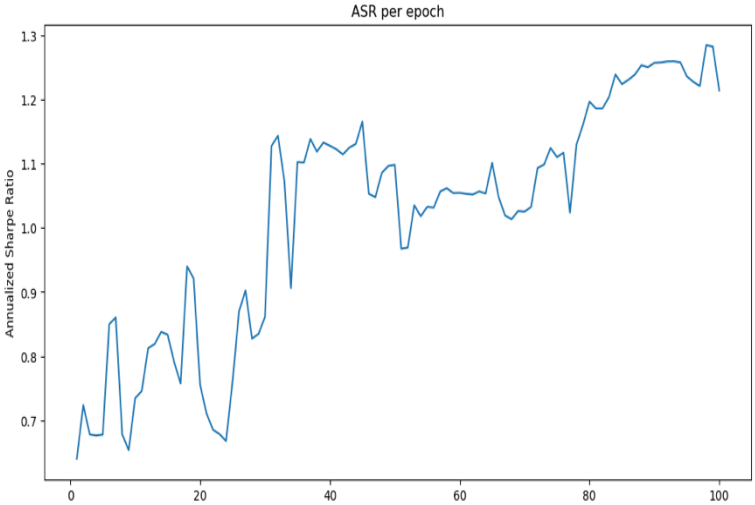

Figure 4.5 - SLP; PCT, V, SMA, Smoothed - Random update

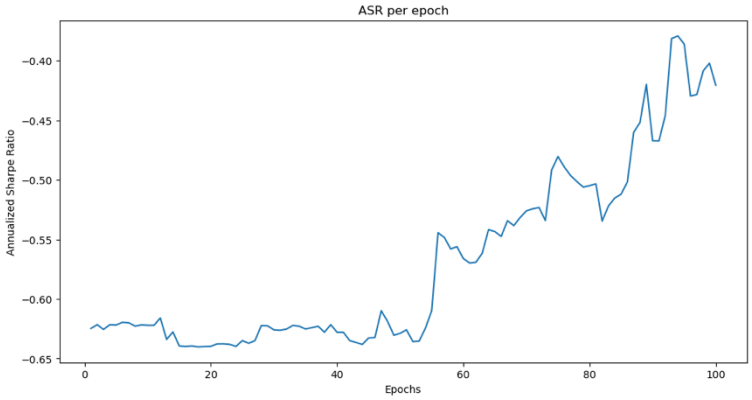

Figure 5 - MLP; PCT - Random Update

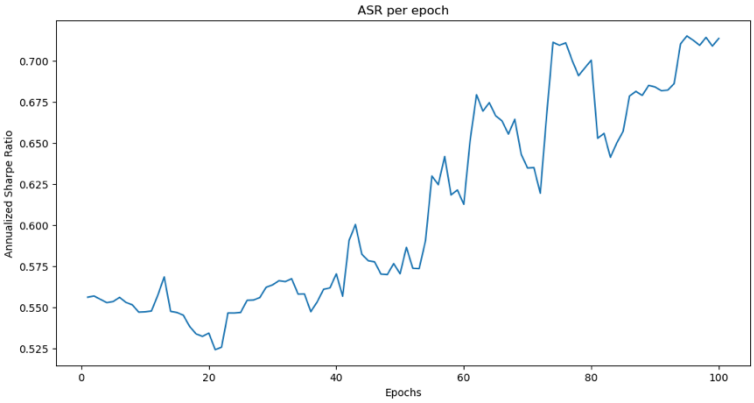

Figure 5.1 - MLP; PCT, EMA - Random update

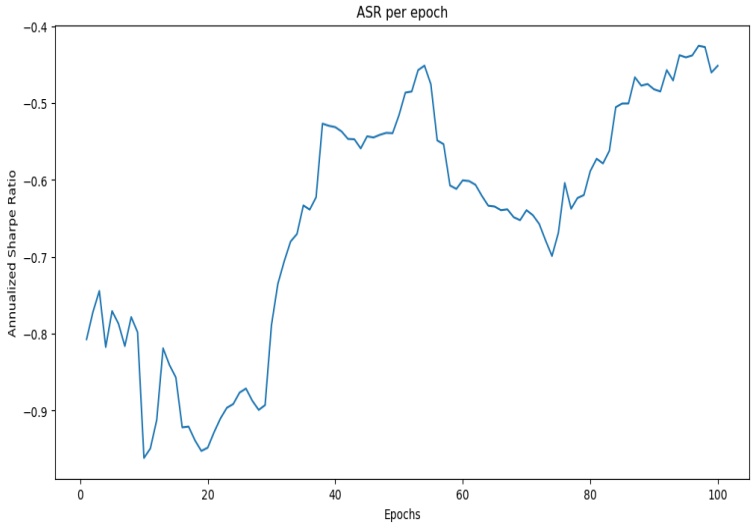

Figure 5.2 - MLP; PCT, V, SMA - Random update

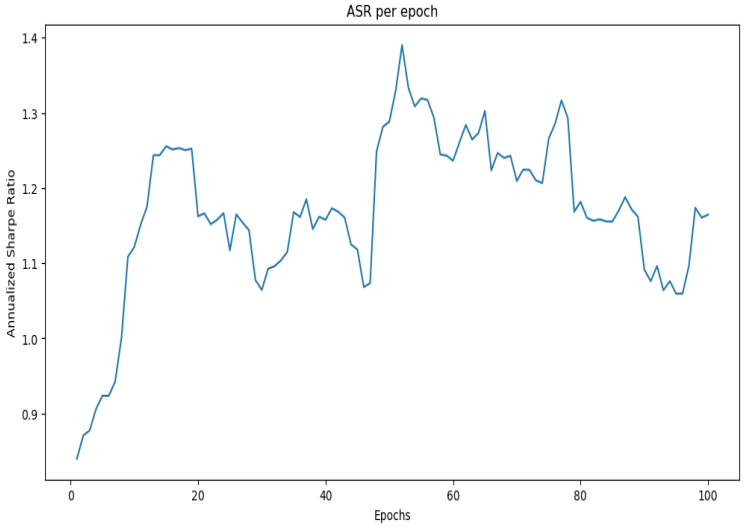

Figure 5.3 - MLP; PCT, Smoothed - Random update

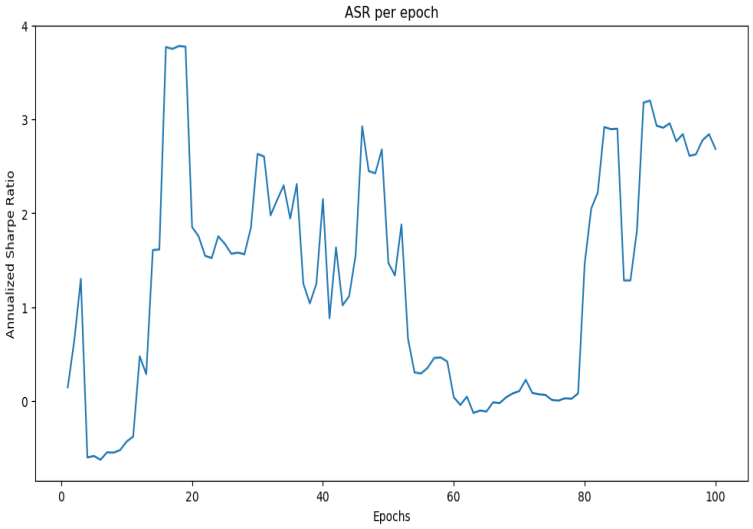

Figure 5.4 - MLP; PCT, EMA, Smoothed - Random update

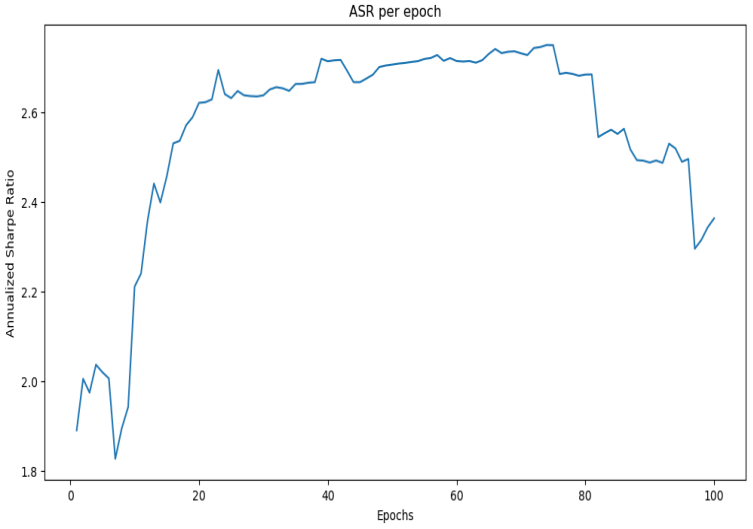

Figure 5.5 - MLP; PCT, V, SMA, Smoothed - Random update

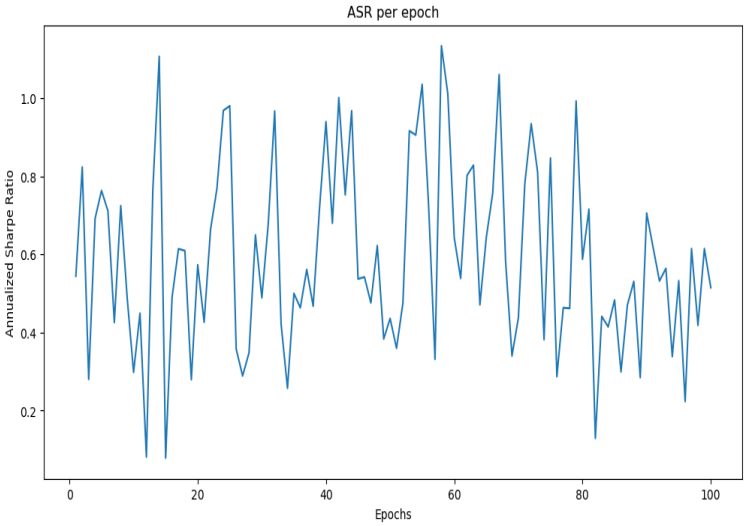

Figure 6 - DLM; PCT - Random update

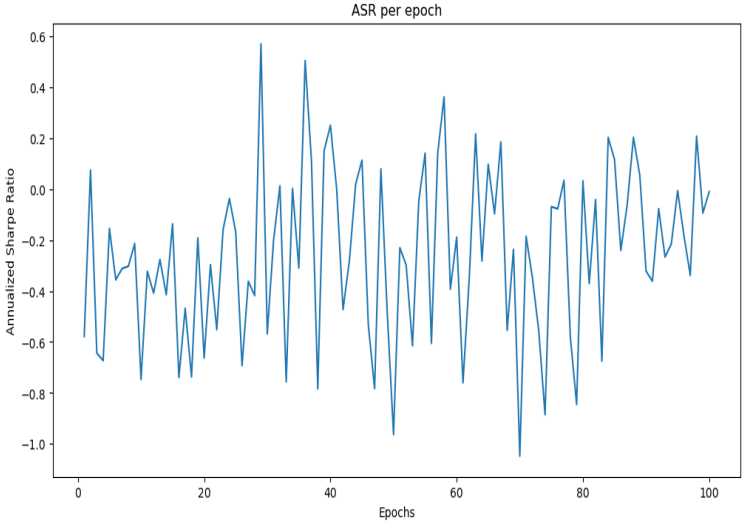

Figure 6.1 - DLM; PCT, EMA - Random update

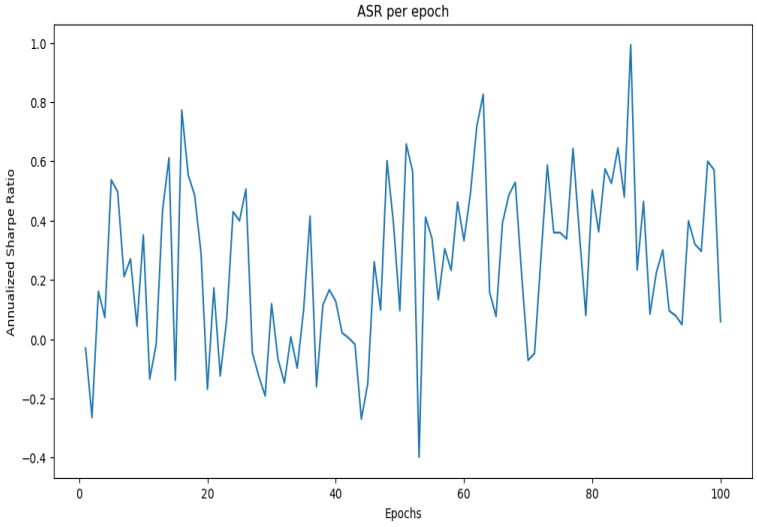

Figure 6.2 - DLM; PCT, V, SMA - Random update

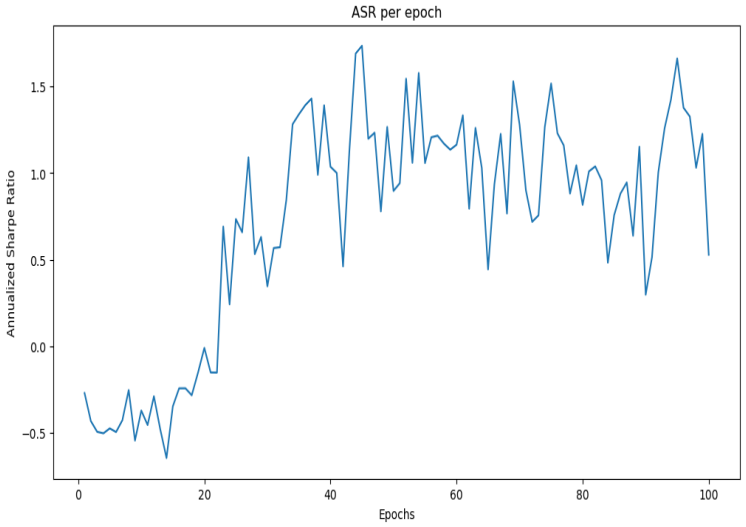

Figure 6.3 - DLM; PCT, Smoothed - Random update

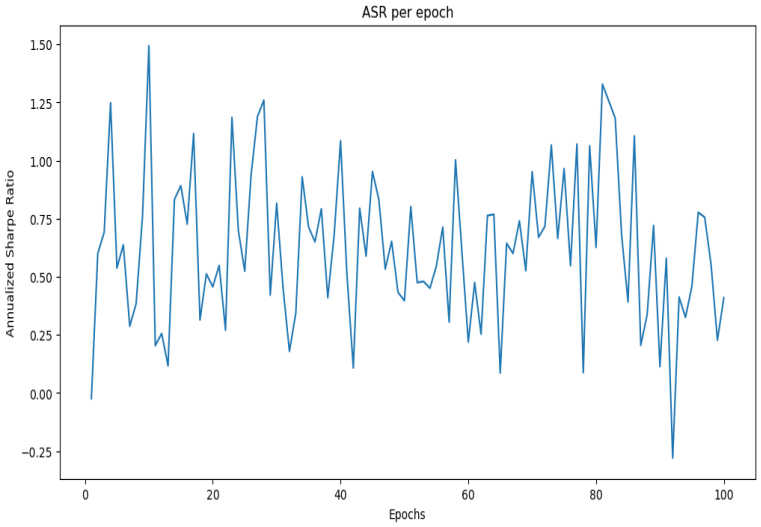

Figure 6.4 - DLM; PCT, EMA, Smoothed - Random update

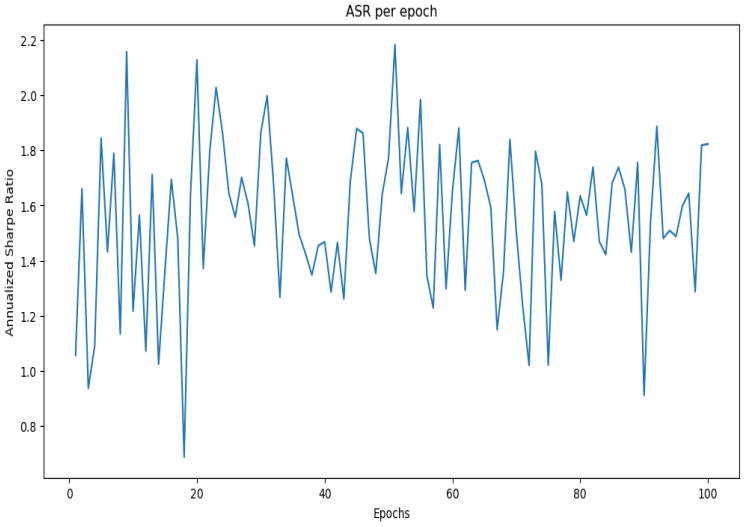

Figure 6.5 - DLM; PCT, V, SMA, Smoothed- Random update

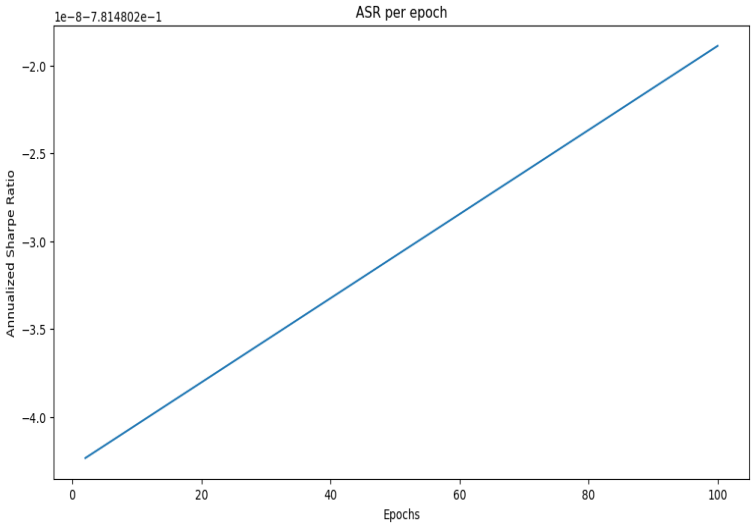

Figure 7 - SLP; PCT - Best update

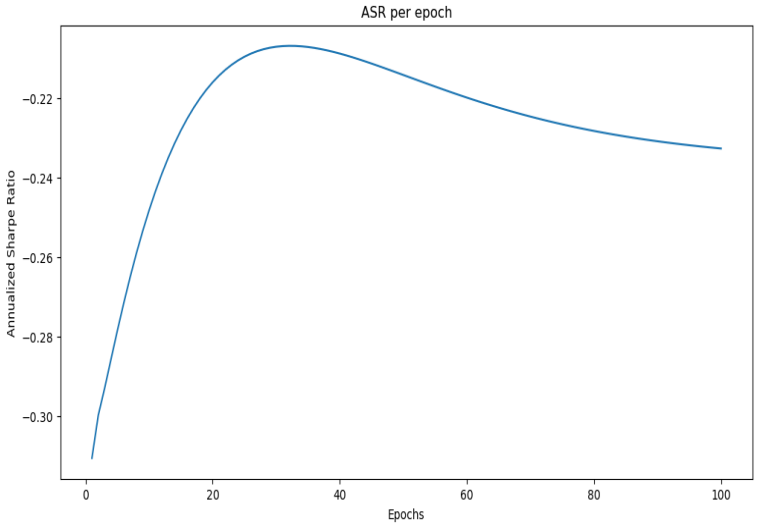

Figure 7.1 - SLP; PCT, EMA - Best update

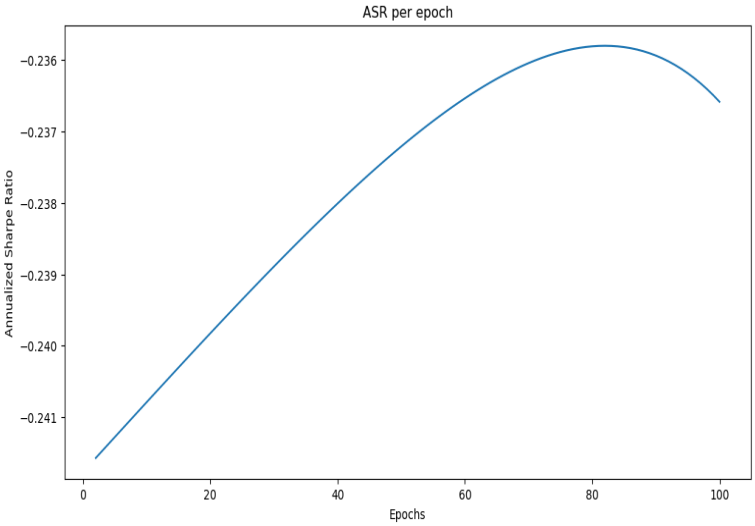

Figure 7.2 - SLP; PCT, V, SMA - Best update

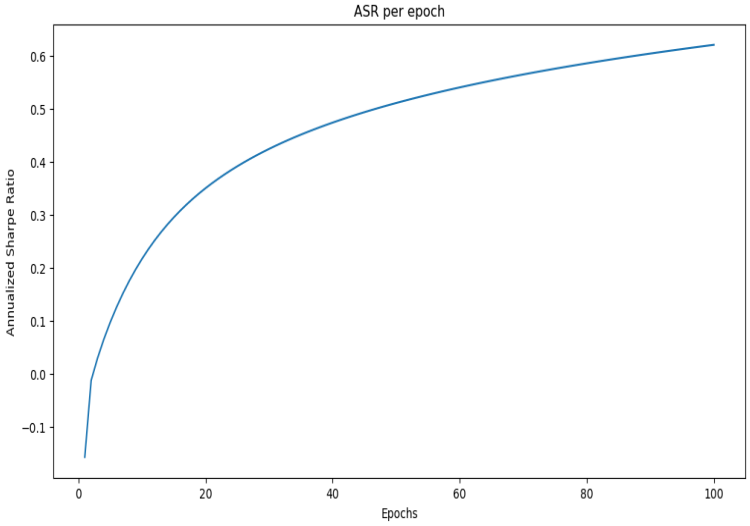

Figure 7.3 - SLP; PCT, Smoothed - Best update

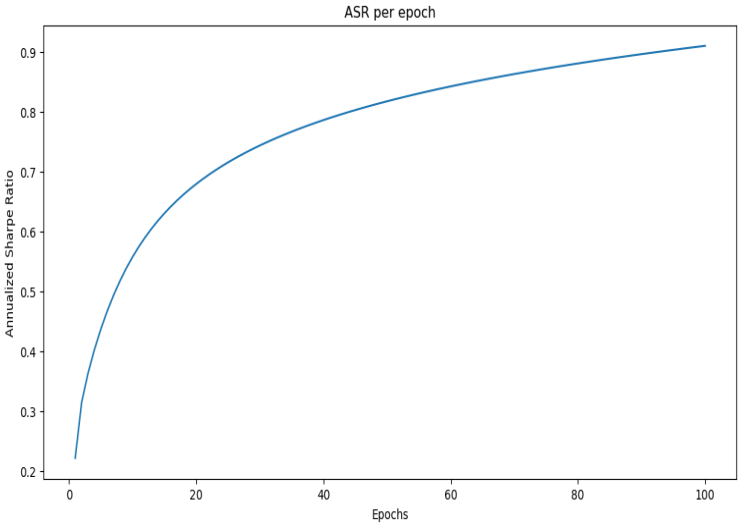

Figure 7.4 - SLP; PCT, EMA, Smoothed - Best update

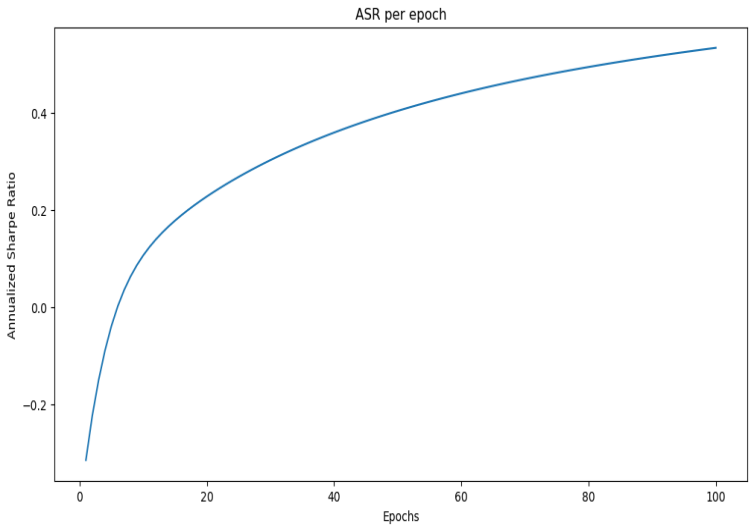

Figure 7.5 - SLP; PCT, V, SMA, Smoothed - Best update

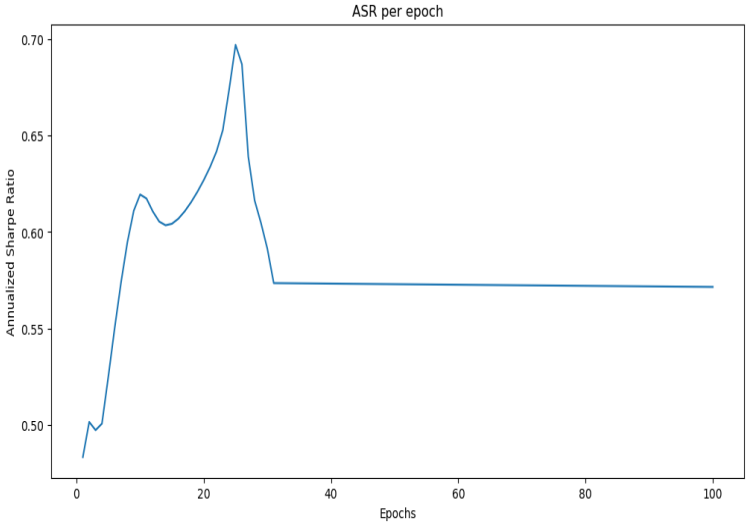

Figure 8 - MLP; PCT - Best update

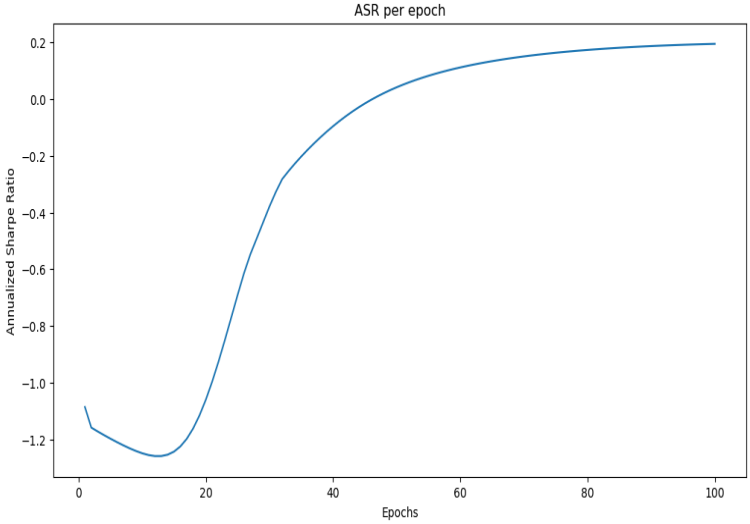

Figure 8.1 – MLP; PCT, EMA - Best update

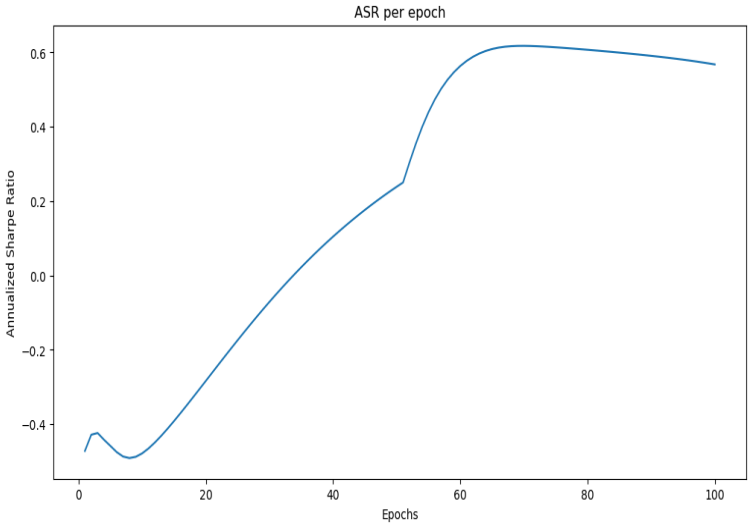

Figure 8.2 - MLP; PCT, V, SMA - Best update

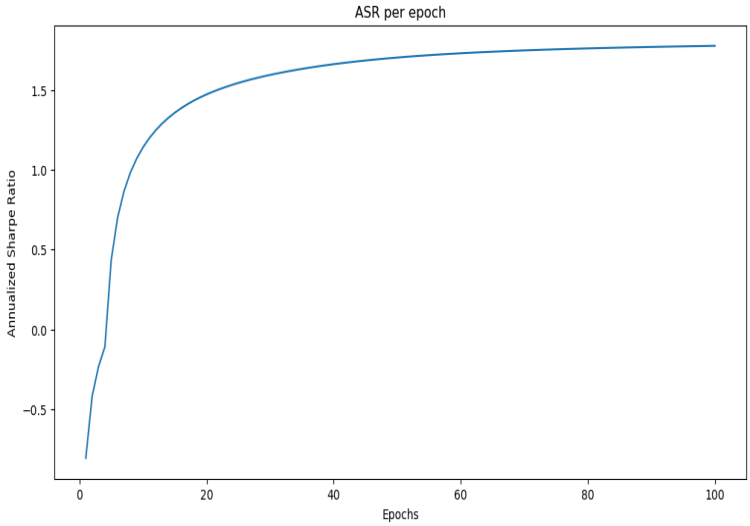

Figure 8.3 - MLP; PCT, Smoothed - Best update

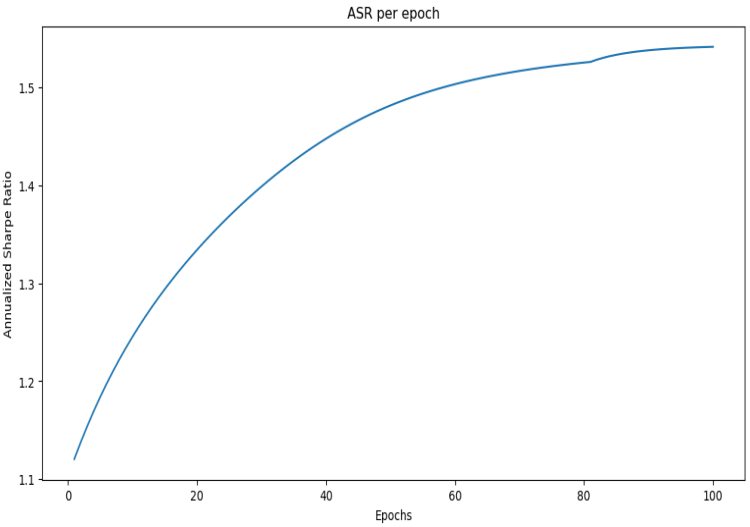

Figure 8.4 - MLP; PCT, EMA, Smoothed - Best update

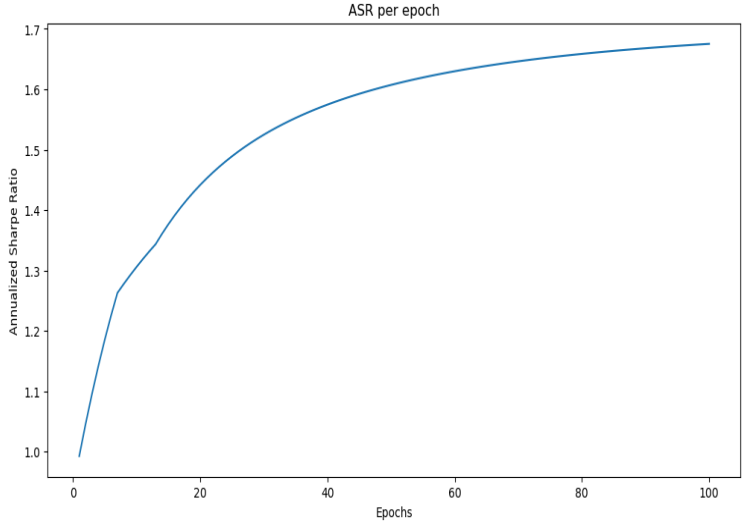

Figure 8.5 - MLP; PCT, V, SMA, Smoothed - Best update

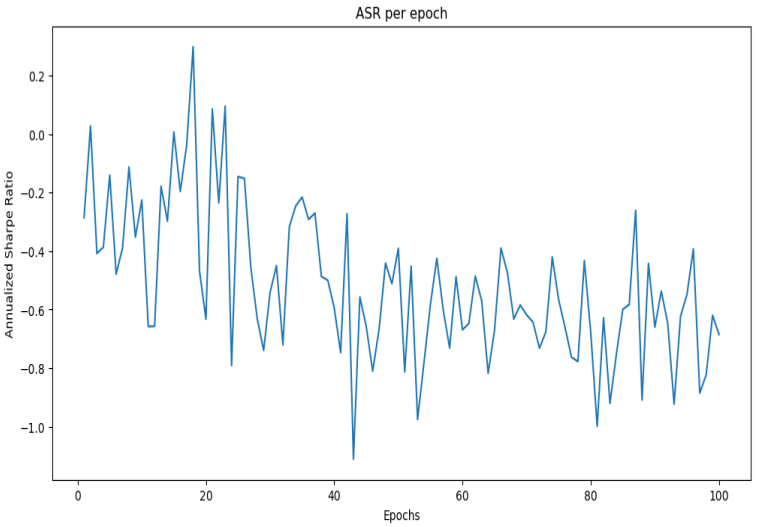

Figure 9 - DLM; PCT - Best update

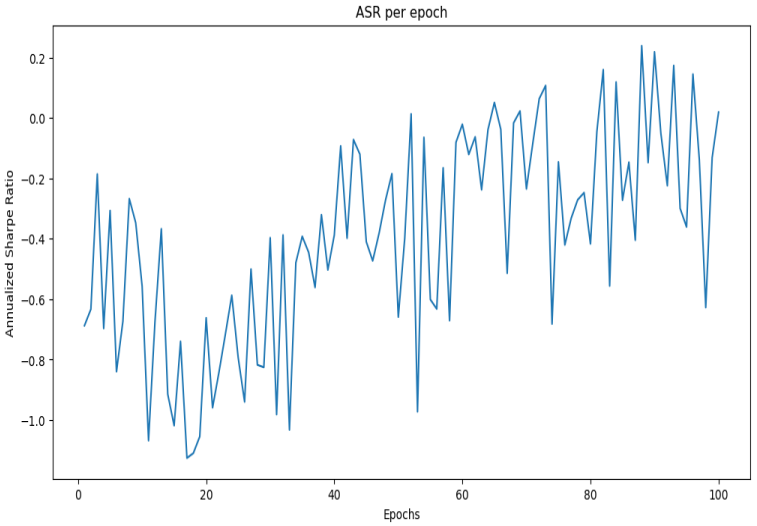

Figure 9.1 - DLM; PCT, EMA - Best update

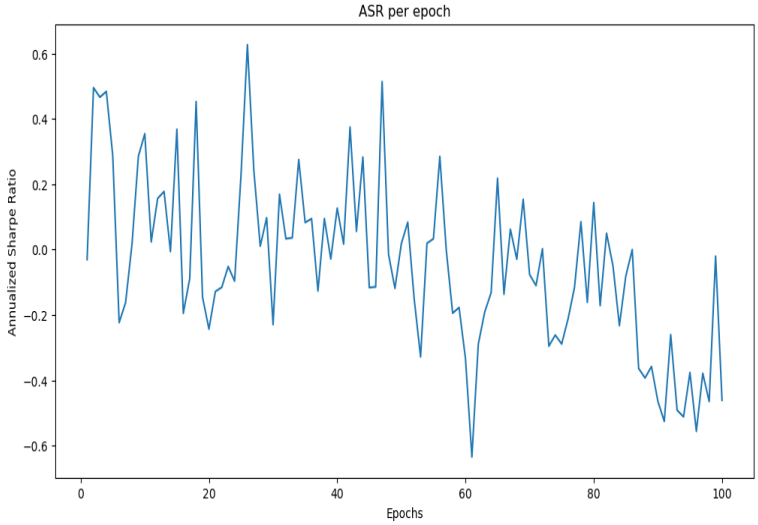

Figure 9.2 - DLM; PCT, V, SMA - Best update

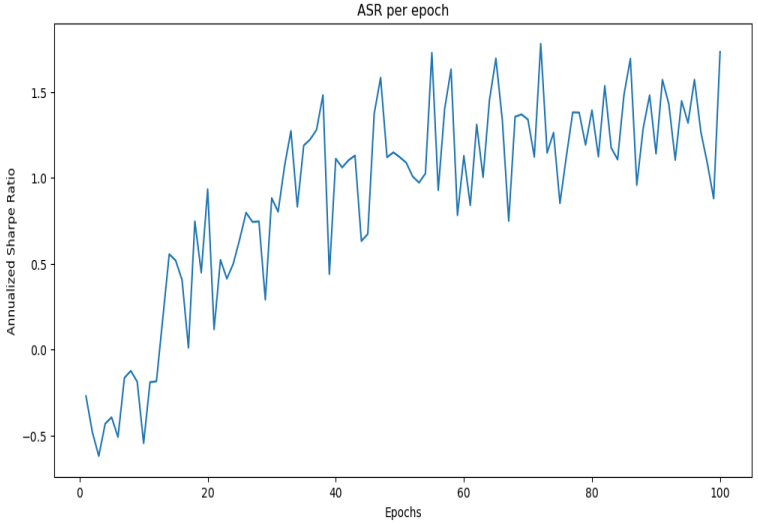

Figure 9.3 - DLM; PCT, Smoothed - Best update

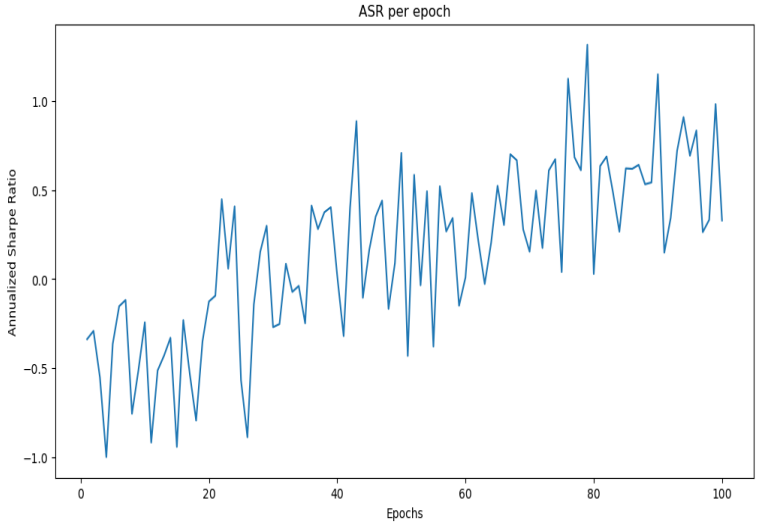

Figure 9.4 - DLM; PCT, EMA, Smoothed - Best update

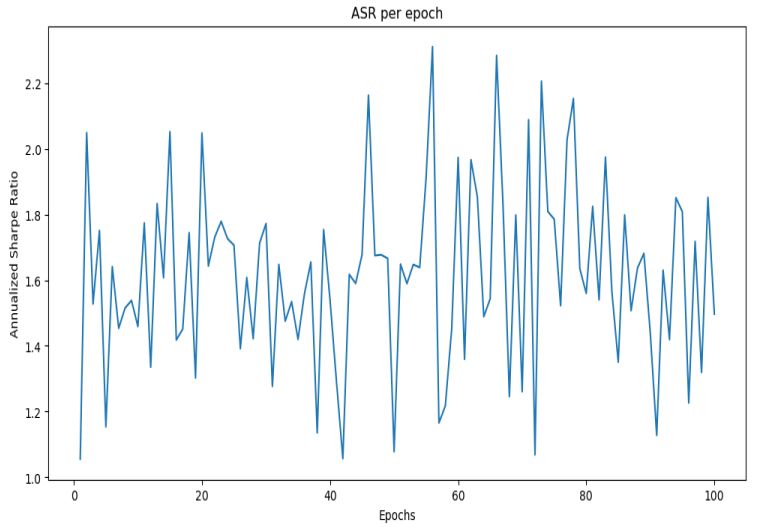

Figure 9.5 - DLM; PCT, V, SMA, Smoothed - Best update

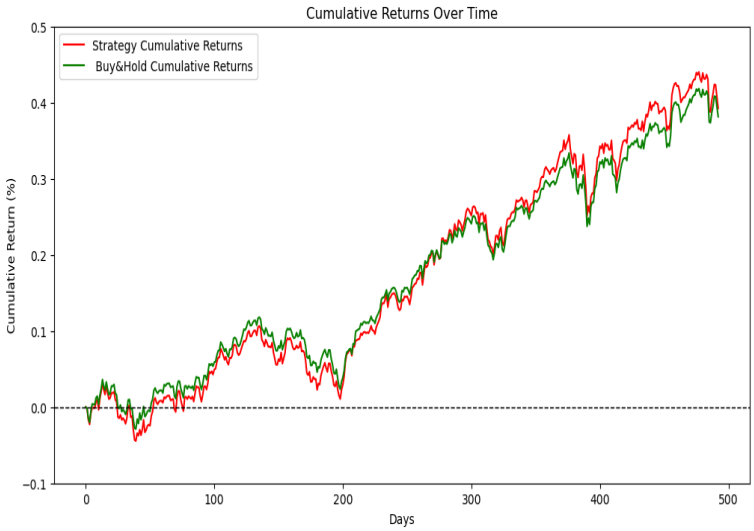

Figure 10 - SLP; PCT - Random update - benchmark comparison

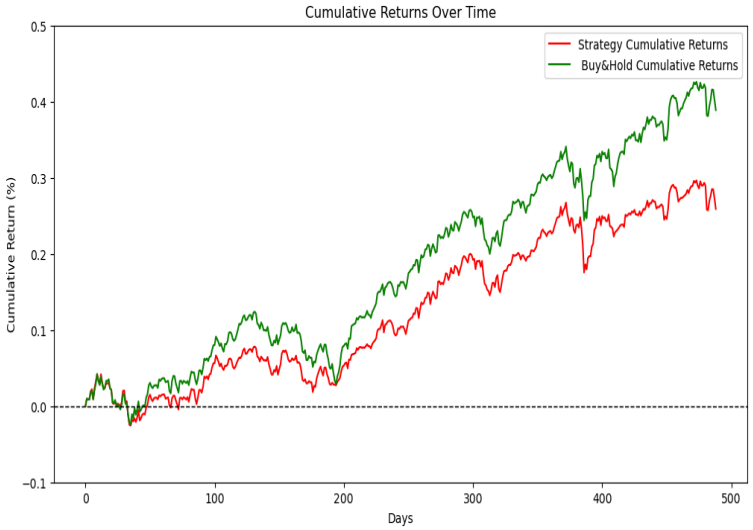

Figure 10.1 - SLP; PCT, EMA - Random update - benchmark comparison

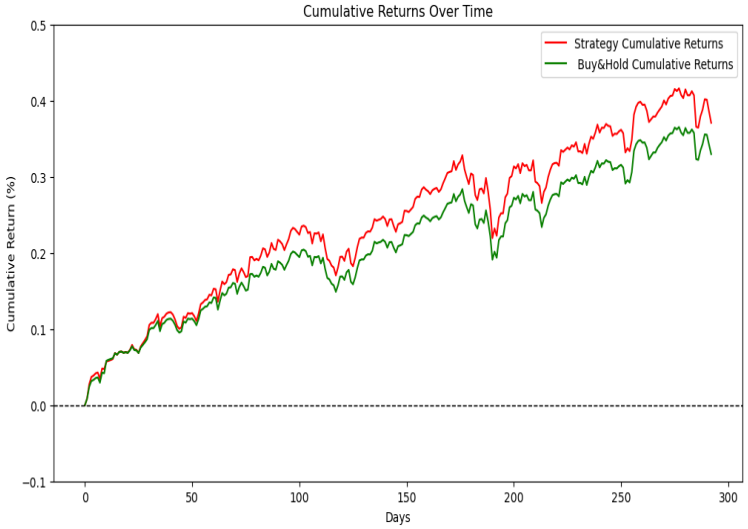

Figure 10.2 - SLP; PCT, V, SMA - Random update - benchmark comparison

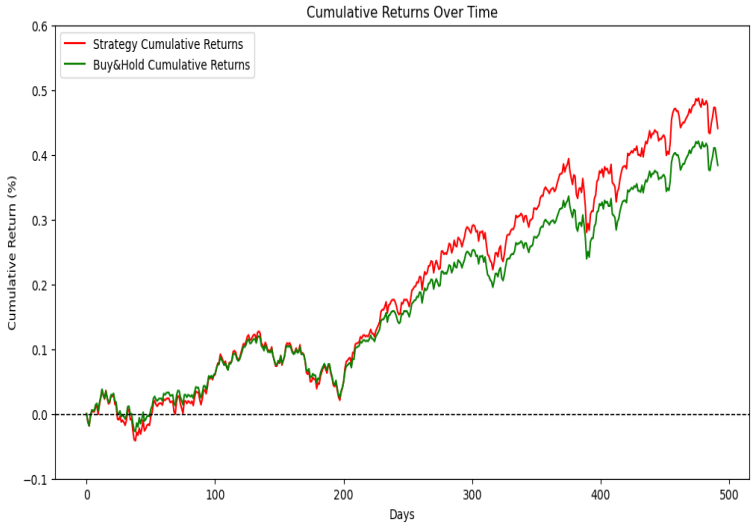

Figure 10.3 - SLP; PCT, Smoothed - Random update - benchmark comparison

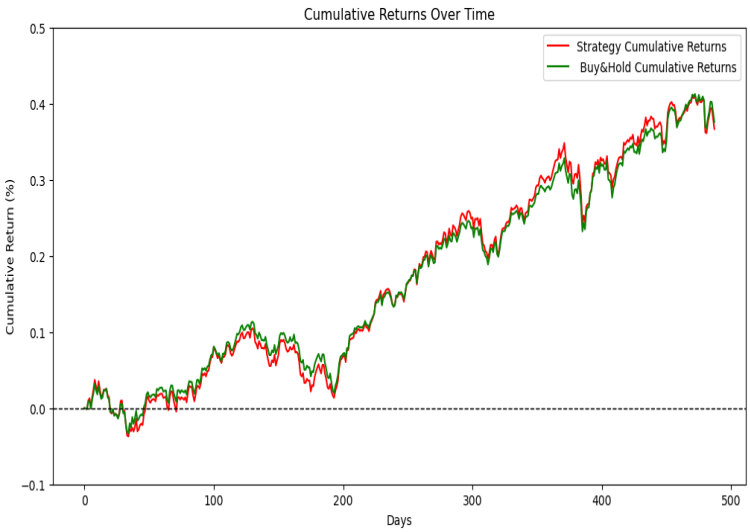

Figure 10.4 - SLP; PCT, EMA, Smoothed - Random update - benchmark comparison

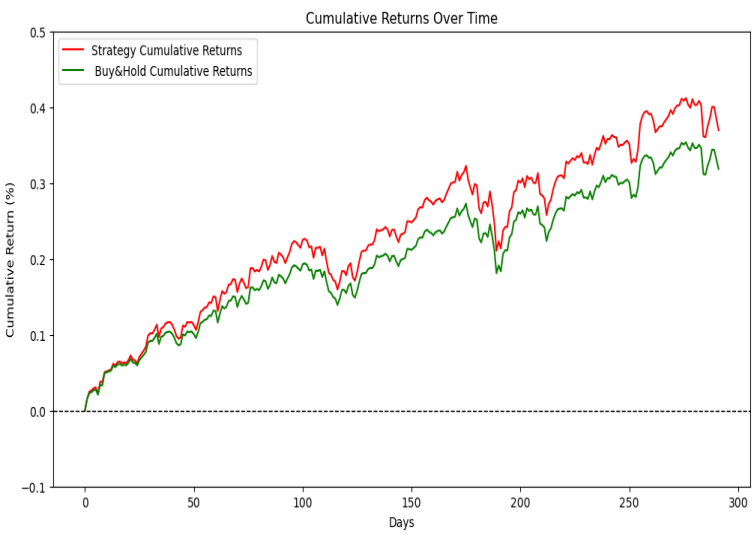

Figure 10.5 - SLP; PCT, V, SMA, Smoothed - Random update - benchmark comparison

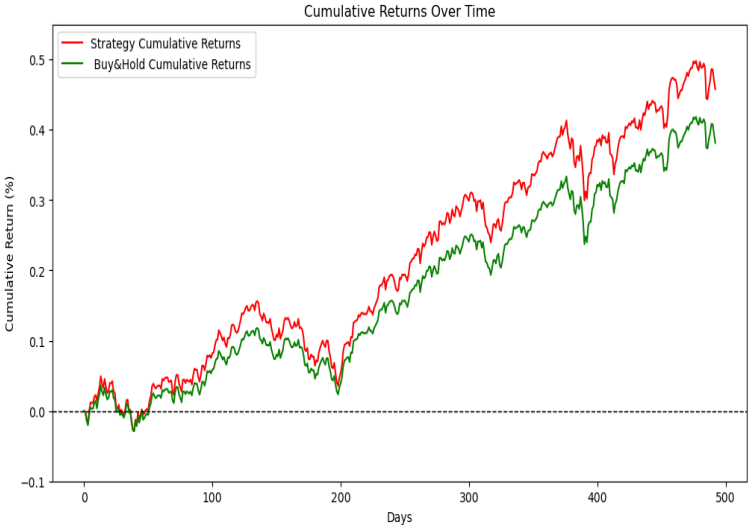

Figure 11 - MLP; PCT - Random update - benchmark comparison

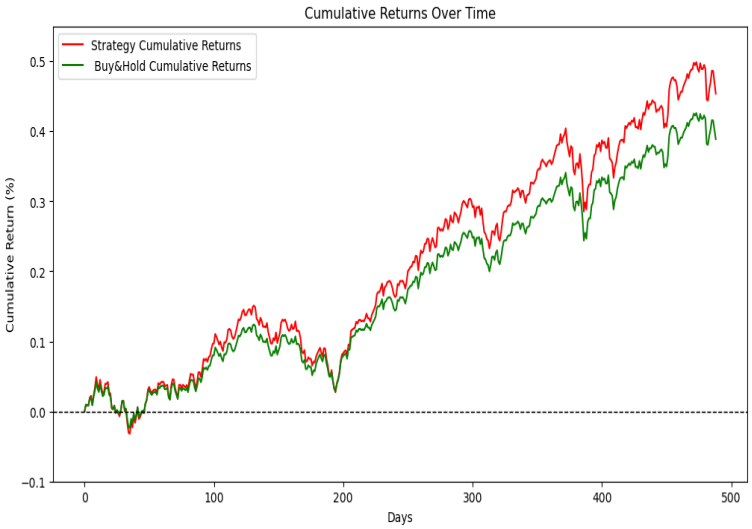

Figure 11.1 - MLP; PCT, EMA - Random update - benchmark comparison

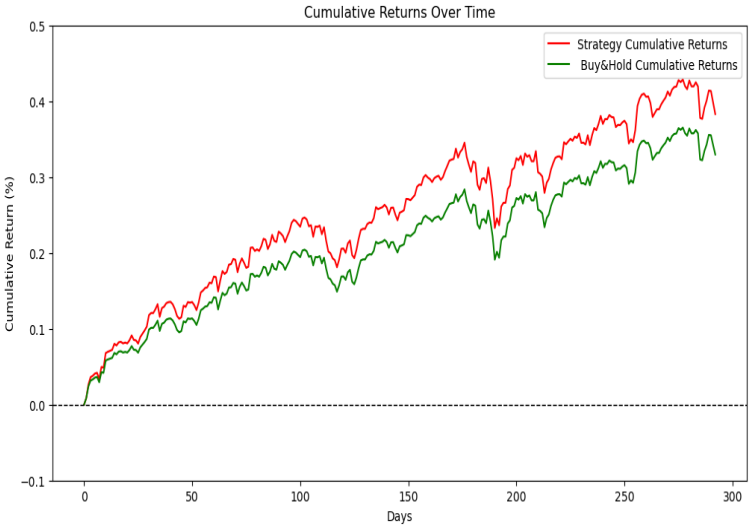

Figure 11.2 - MLP; PCT, V, SMA - Random update - benchmark comparison

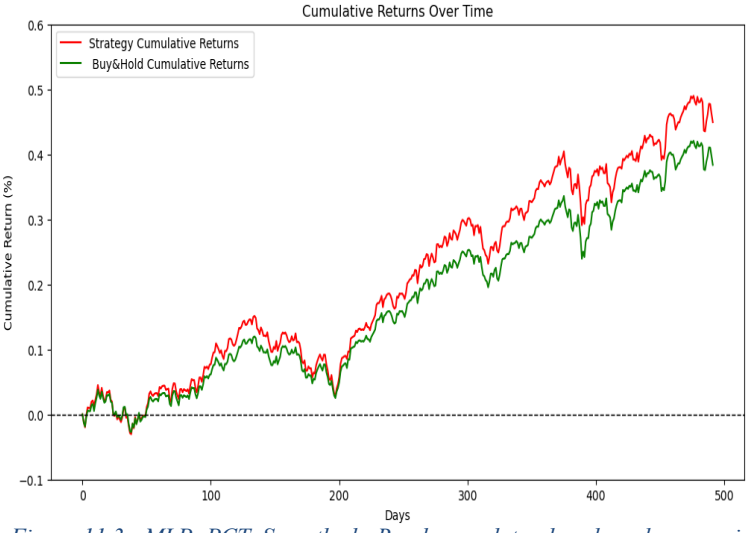

Figure 11.3 - MLP; PCT, Smoothed - Random update - benchmark comparison

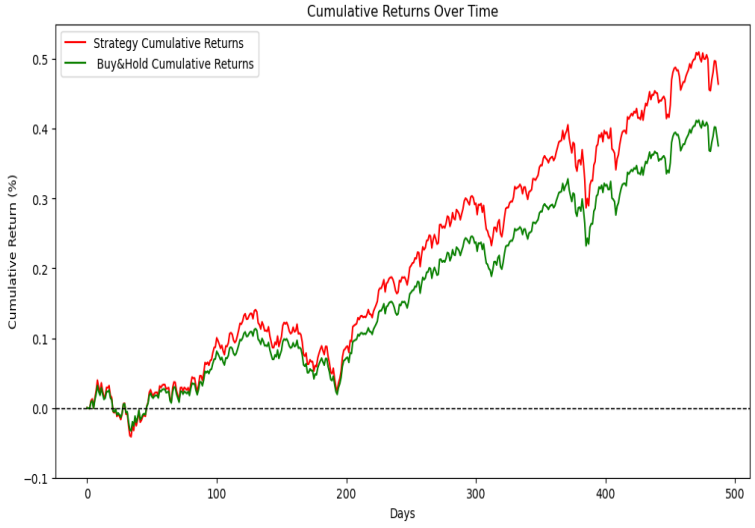

Figure 11.4 - MLP; PCT, EMA, Smoothed - Random update - benchmark comparison

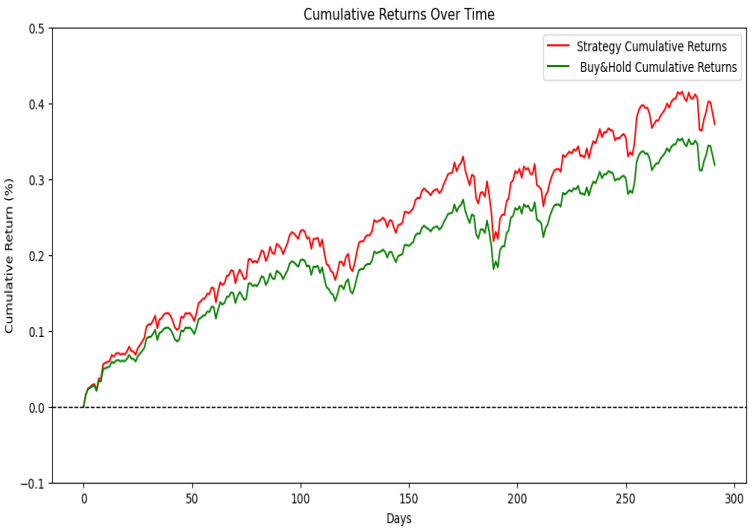

Figure 11.5 - MLP; PCT, V, SMA, Smoothed - Random update - benchmark comparison

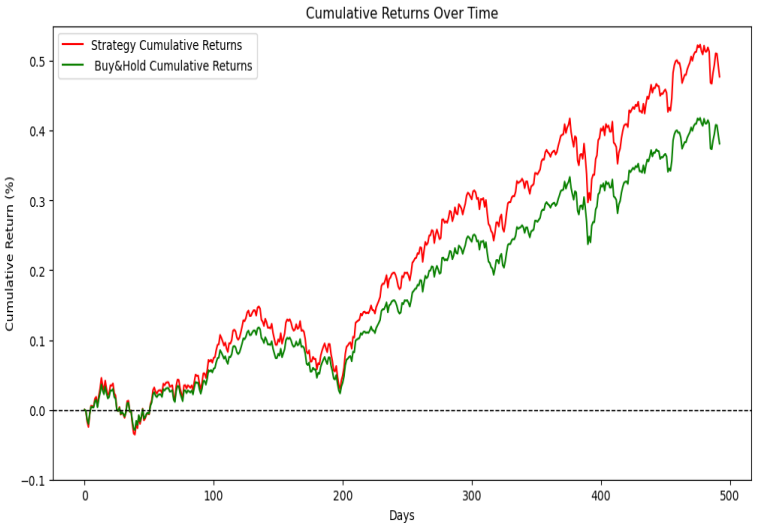

Figure 12 - DLM; PCT - Random update - benchmark comparison

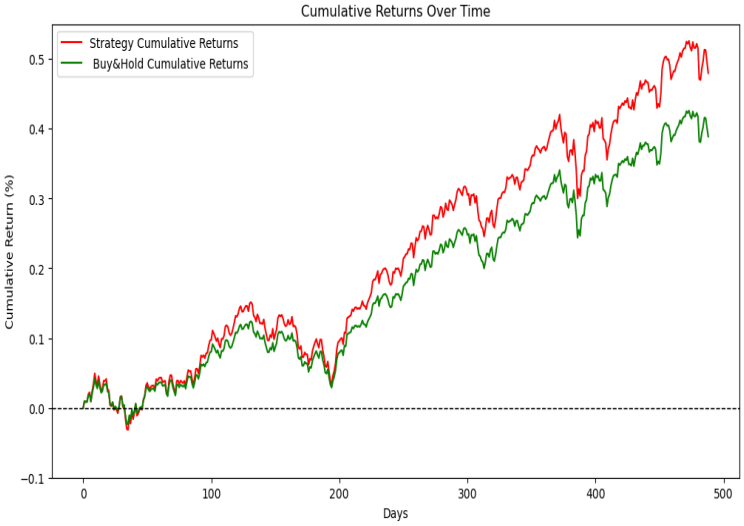

Figure 12.1 - DLM; PCT, EMA - Random update - benchmark comparison

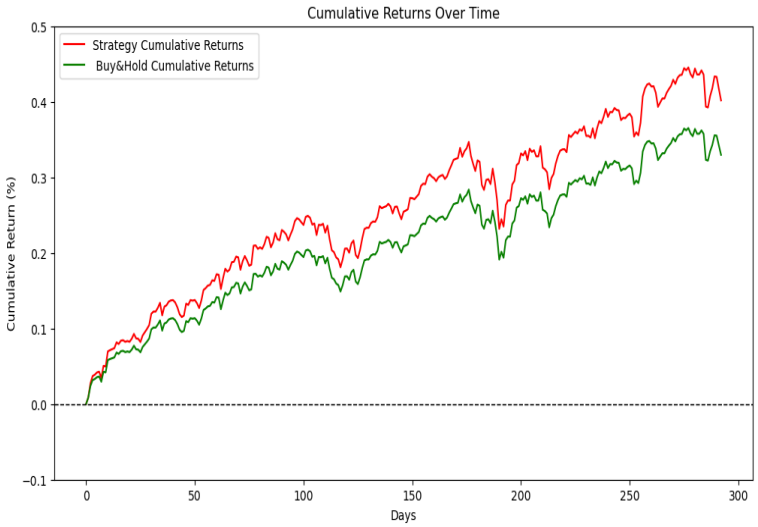

Figure 12.2 - DLM; PCT, SMA, V - Random update - benchmark comparison

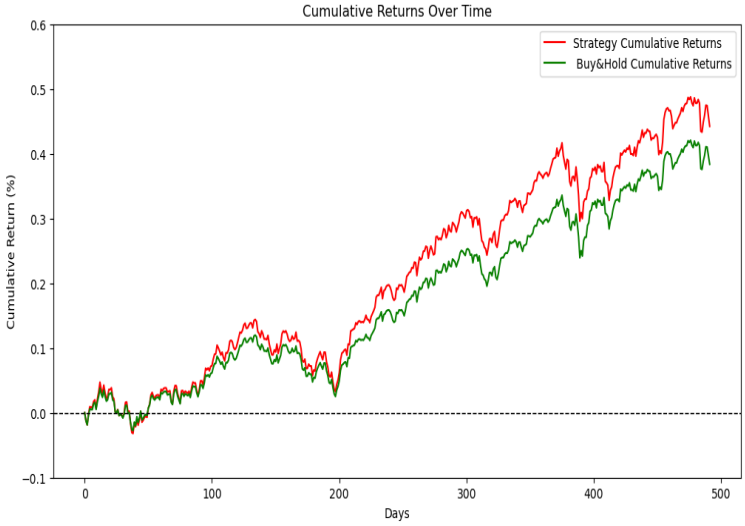

Figure 12.3 - DLM; PCT, Smoothed - Random update - benchmark comparison

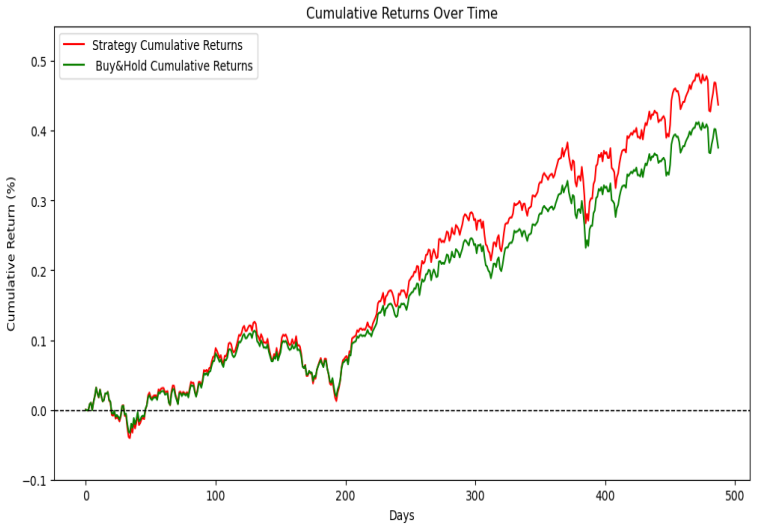

Figure 12.4 - DLM; PCT, EMA, Smoothed - Random update - benchmark comparison

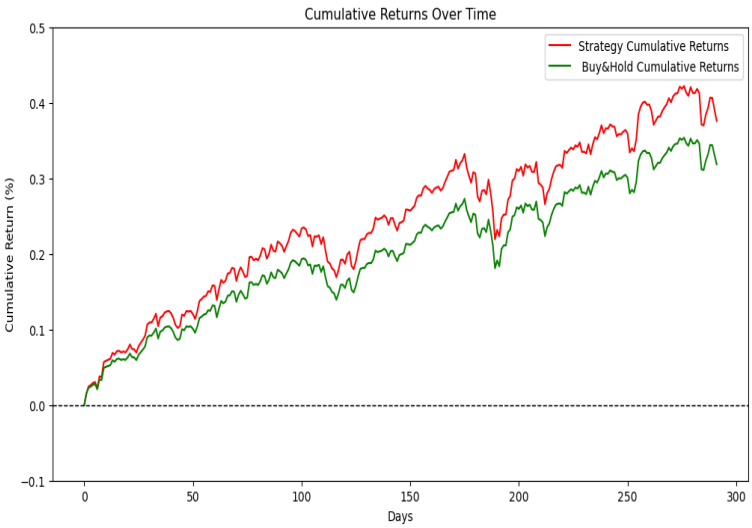

Figure 12.5 - DLM; PCT, V, SMA, Smoothed - Random update - benchmark comparison

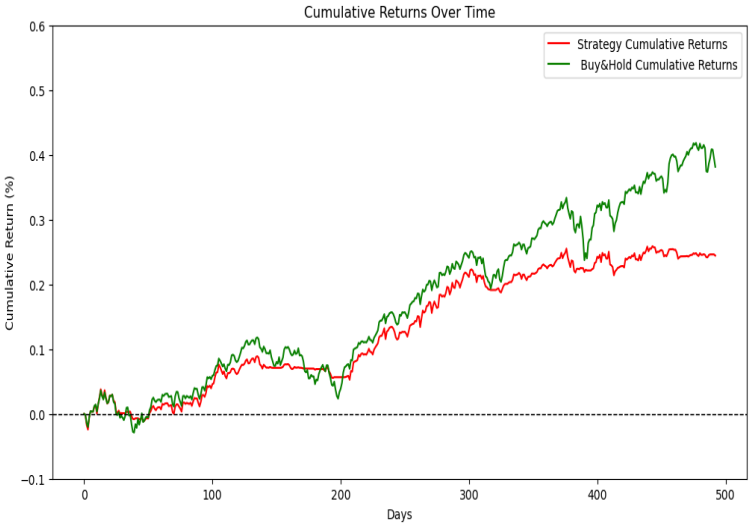

Figure 13 - SLP; PCT - Best update - benchmark comparison

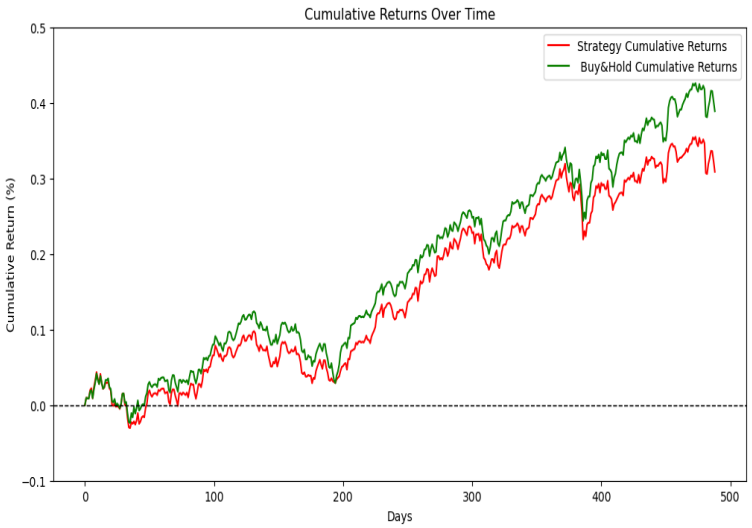

Figure 13.1 - SLP; PCT, EMA - Best update - benchmark comparison

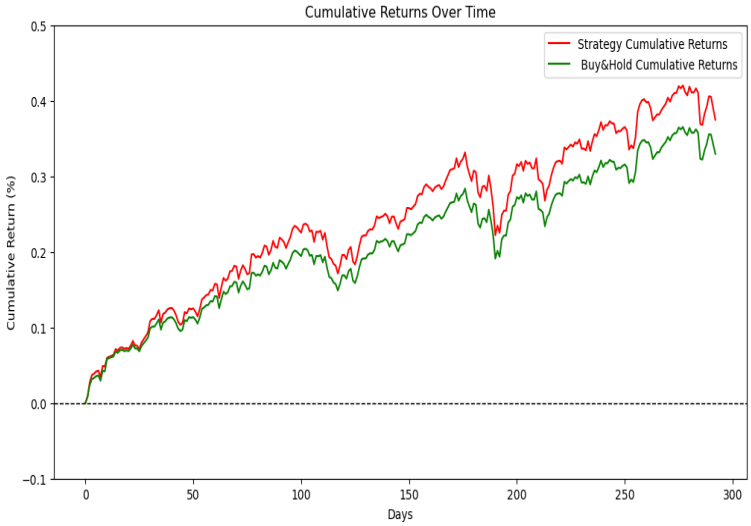

Figure 13.2 - SLP; PCT, V, SMA - Best update - benchmark comparison

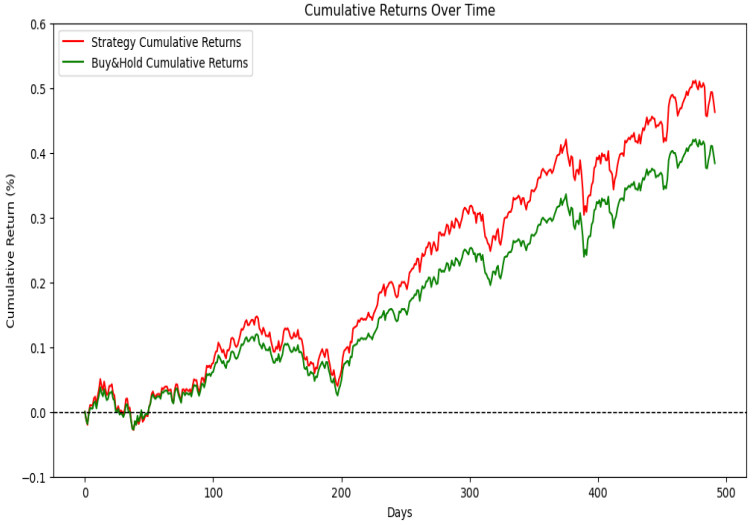

Figure 13.3 - SLP; PCT, Smoothed - Best update - benchmark comparison

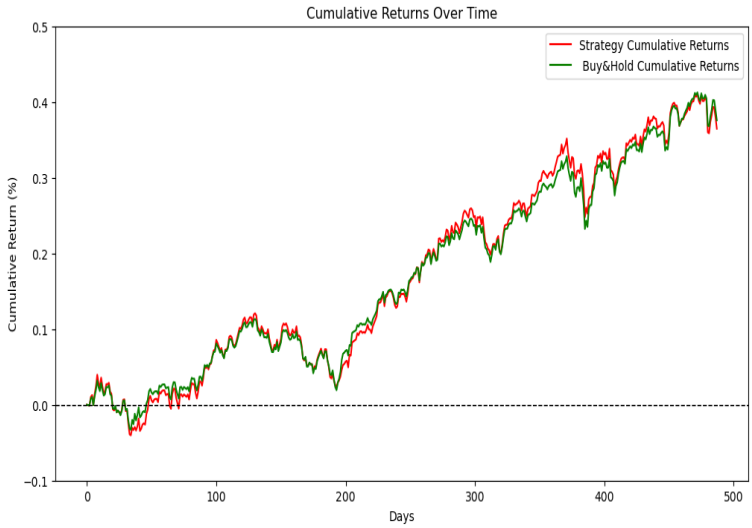

Figure 13.4 - SLP; PCT, EMA, Smoothed - Best update - benchmark comparison

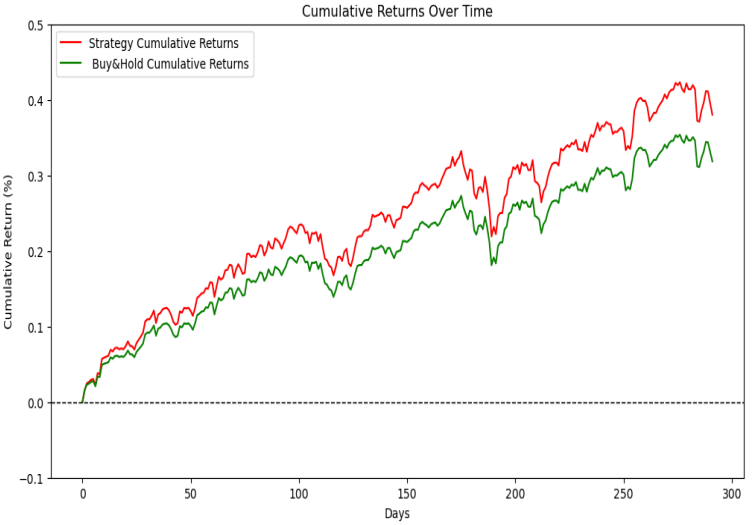

Figure 13.5 - SLP; PCT, V, SMA, Smoothed - Best update - benchmark comparison

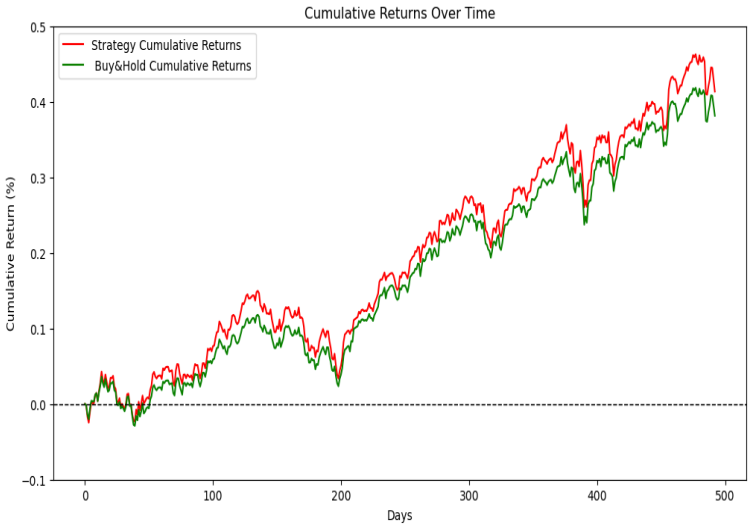

Figure 14 - MLP; PCT - Best update - benchmark comparison

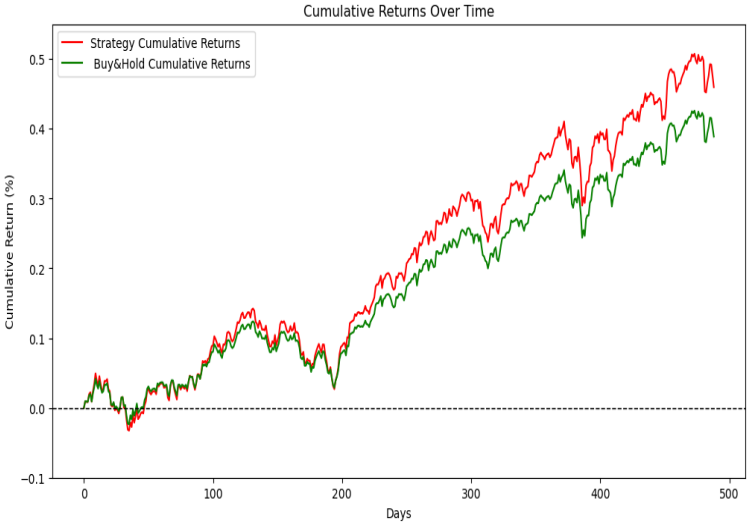

Figure 14.1 - MLP; PCT, EMA - Best update - benchmark comparison

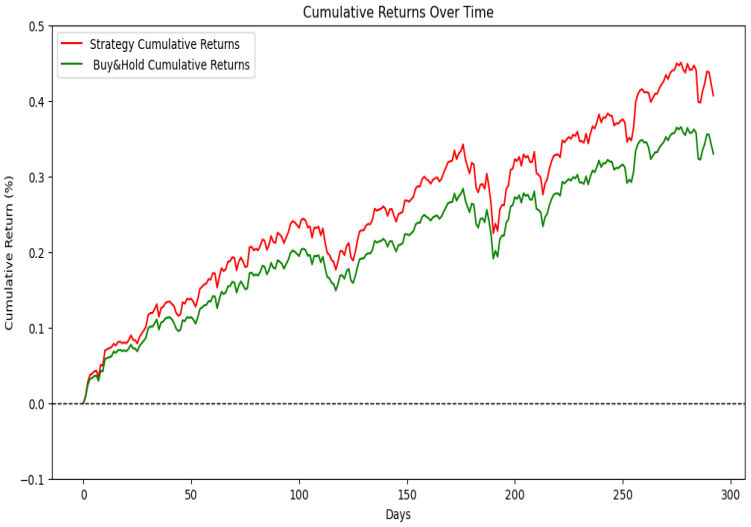

Figure 14.2 - MLP; PCT, V, SMA - Best update - benchmark comparison

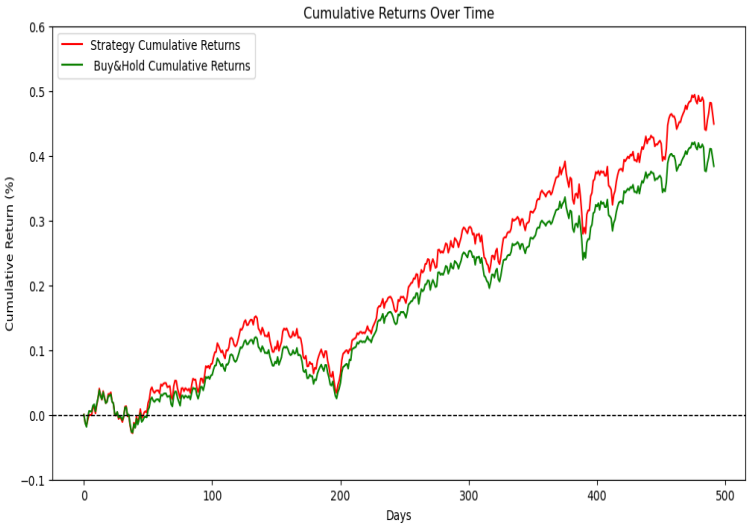

Figure 14.3 - MLP; PCT, Smoothed - Best update - benchmark comparison

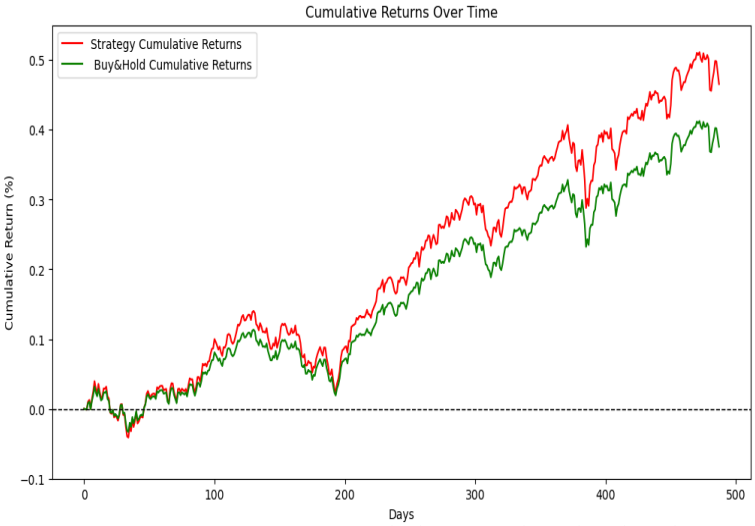

Figure 14.5 - MLP; PCT, EMA, Smoothed - Best update - benchmark comparison

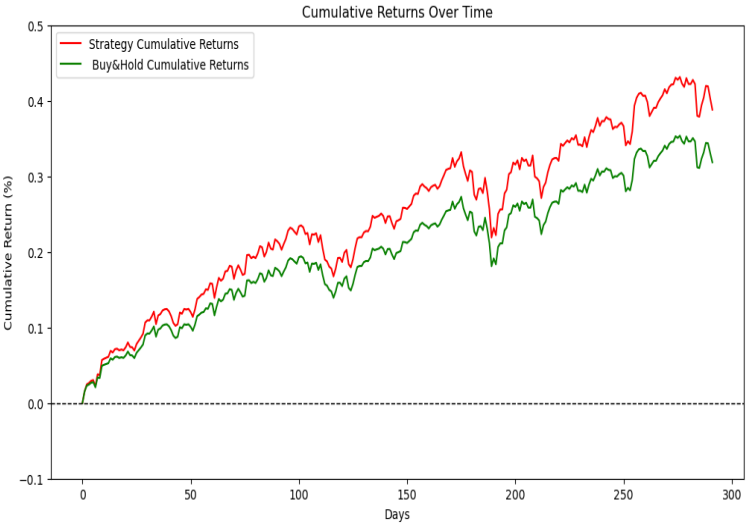

Figure 14.4 - MLP; PCT, V, SMA, Smoothed - Best update - benchmark comparison

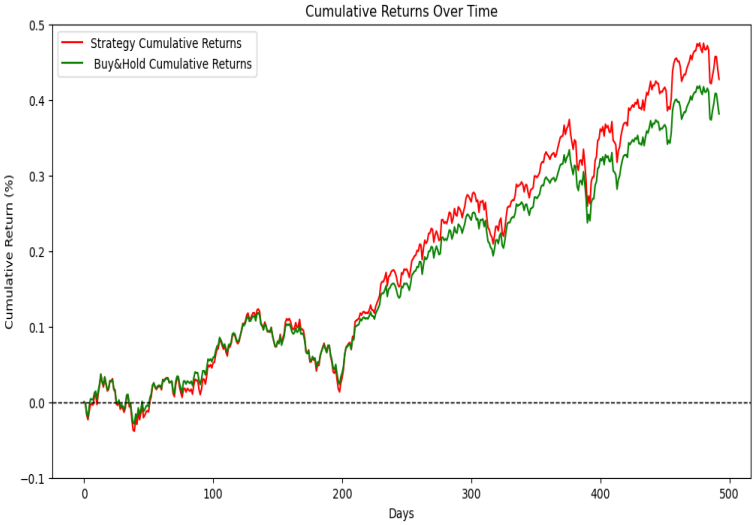

Figure 15 - DLM; PCT - Best update - benchmark comparison

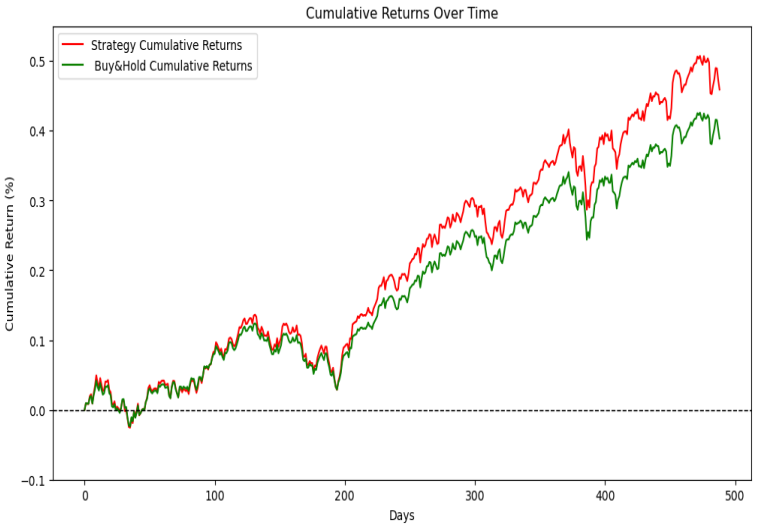

Figure 15.1 - DLM; PCT, EMA - Best update - benchmark comparison

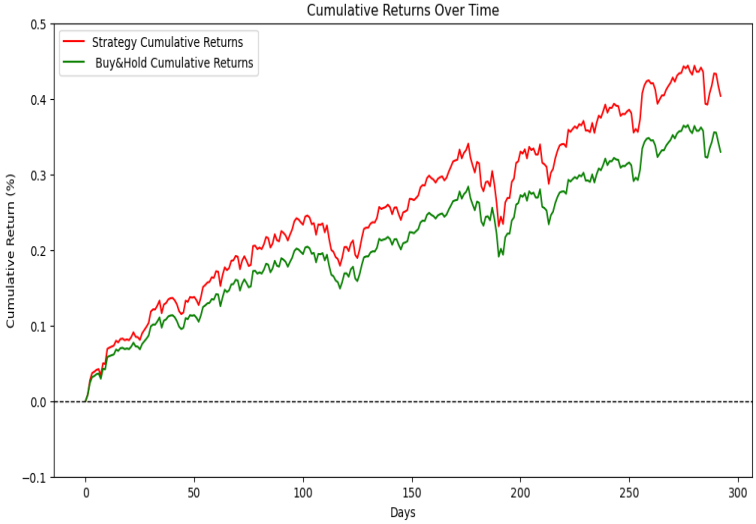

Figure 15.2 - DLM; PCT, V, SMA - Best update - benchmark comparison

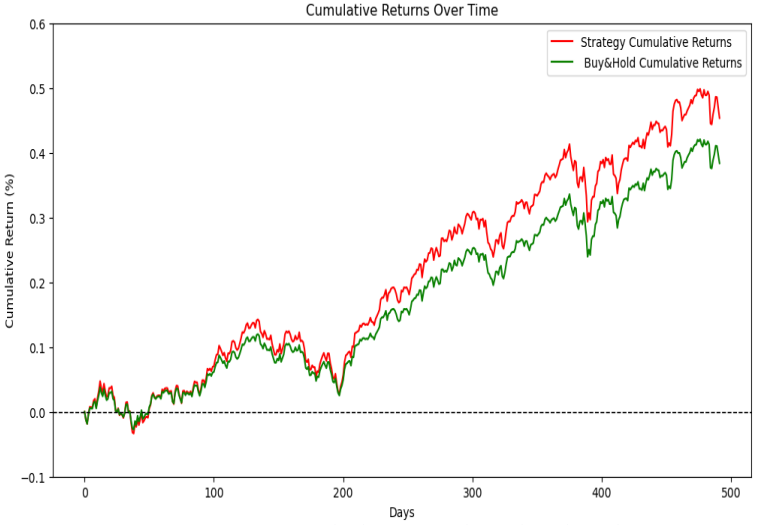

Figure 15.3 - DLM; PCT, Smoothed - Best update - benchmark comparison

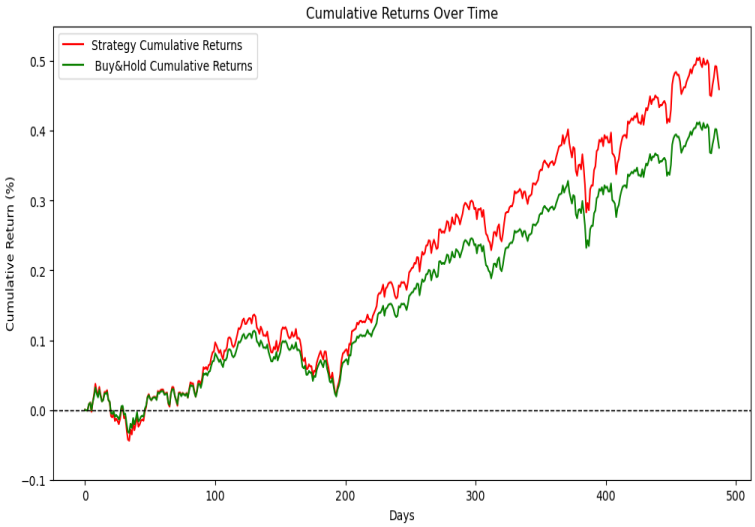

Figure 15.4 - DLM; PCT, EMA, Smoothed - Best update - benchmark comparison

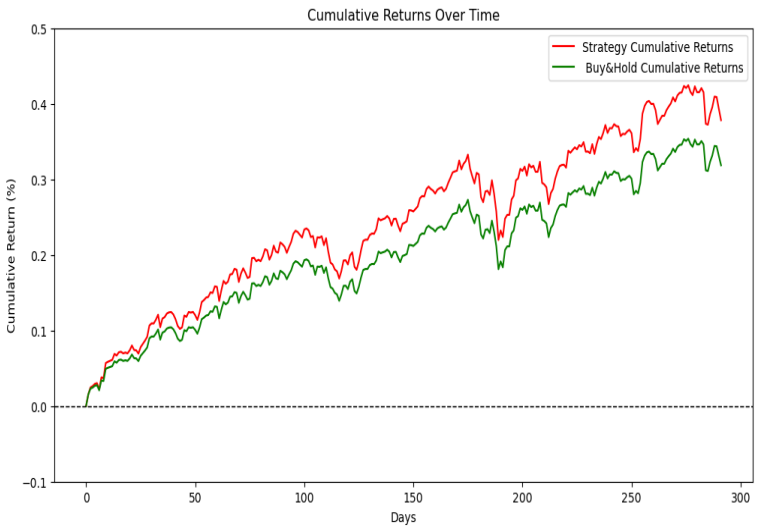

Figure 15.5 - DLM; PCT, V, SMA, Smoothed - Best update - benchmark comparison

Bibliography

[1] – Sharpe Ratio: https://www.investopedia.com/terms/s/sharperatio.asp
[2] – *Neural Networks, A Comprehensive Foundation; Simon Haykin*
[3] – *Reinforcement Learning for Trading; Moody and Saffell, 1998*
[4] – *Reinforcement Learning for Systematic FX trading; Borrageiro, Firoozye and Barucca, 2021*
[5] – *Nonlinear Trading Models Through Sharpe Ratio Maximization; Choey and Weigand, 1997*
[6] – *Deep Learning for Financial Applications: A Survey; Ozbayoglu, Gudelek and Sezer*
[7] – *Using a Financial Criterion Rather Than a Prediction Criterion; Bengio*
[8] – Min-Max Normalisation: https://en.wikipedia.org/wiki/Feature_scaling
[9] – Standardisation: https://en.wikipedia.org/wiki/Feature_scaling
[10] – tanh: https://en.wikipedia.org/wiki/Hyperbolic_functions
[11] – Pandas: https://pandas.pydata.org
[12] – NumPy: https://numpy.org
[13] – Matplotlib: https://matplotlib.org
[14] – Yahoo Finance: https://uk.finance.yahoo.com
[15] – Alternative Trading Strategies: https://en.wikipedia.org/wiki/Trading_strategy
[16] – Pruning: https://en.wikipedia.org/wiki/Pruning_(artificial_neural_network)
[17] – Alternative Assets: https://www.investopedia.com/terms/a/assetclasses.asp
[18] – Recurrent Neural Network: https://en.wikipedia.org/wiki/Recurrent_neural_network
[19] – Long Short Term Memory: https://en.wikipedia.org/wiki/Long_short-term_memory
[20] – Convolutional Neural Networks: https://en.wikipedia.org/wiki/Convolutional_neural_network
[21] – Q Learning: https://en.wikipedia.org/wiki/Q-learning
[22] – Sentiment Analysis: https://en.wikipedia.org/wiki/Sentiment_analysis
[23] – Macroeconomic Indicators: https://www.investopedia.com/terms/e/economic_indicator.asp
[24] – Alternative Technical Indicators: https://www.investopedia.com/terms/t/technicalindicator.asp
[25] – Maximum Drawdown: https://www.investopedia.com/terms/m/maximum-drawdown-mdd.asp
[26] – Calmar/Sortino Ratio: https://medium.com/@williamchristianfowls/sharpe-ratio-sortino-ratio-and-calmar-ratio-4738a3574fdb#
[27] – Transaction Cost (Slippage): https://www.fefundinfo.com/insights/slippage-methodology-navigating-evolving-transaction-cost-requirements