28/10/2025

**INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE**

ITESO

Universidad Jesuita
de Guadalajara

Microstructure and Trading Systems :

# “003 Advanced Trading Strategies: Deep Learning”


Presented by:

Mauricio Martínez Ulloa / 746331
David Campos Ambriz / 744435

Professor:
Luis Felipe Gómez Estrada

# Contents

## Introduction:

For the development of this project, a systematic trading strategy was required to be designed. In this strategy, Deep Learning models are implemented, trained using a time series with the addition of certain technical analysis indicators, to predict buy, sell, and hold signals for each moment. This can be translated as a multi-class classification (outputs 0, 1, and 2) that the models can predict, ultimately converting the output to this desired range through the SoftMax function.

The objective of this project is to obtain signals to operate a portfolio based on one asset, which must have a length of 15 years of daily data. This data is divided into 60% for training, 20% for testing, and 20% for validation, all in chronological order, leaving the first block for the oldest data and the last for the most recent data. The monitoring and visualization of model metrics and performance are carried out using the MLFlow library, which facilitates access to an experiment, as well as each run, with all the selected metrics needed to choose the best models and parameters.

As part of the monitoring, data drift is reviewed; that is, the change in the distributions of the data used to train the model and the data used to evaluate the model's performance is monitored. This is done to observe if there are significant changes in the distributions that prevent the model from operating correctly, forcing the model to be retrained to find a better version and repeat the project workflow.

After selecting the best models, it is necessary to evaluate the model's performance in predicting signals, as well as its operation in a portfolio to trade the selected asset, using backtest.py. This allows for the evaluation of the requested metrics, such as Calmar, Win Rate, Sharpe, etc. The access point where the GitHub repository can be consulted to view all the code progress, and to visualize the results shown by executing main.py, is the following:

https://github.com/ulloa09/Proyecto3_TensorFlow

# Development:

## Asset Analysis:

Initially, it was decided to opt for an atypical stock, Wynn Resorts. This is a stock that does not have as high a volume as technology companies, nor does it have high relevance in the market by being a significant part of the portfolios of major stock market competitors. However, it possesses a global presence with significant renown in the hospitality sector.

Data from the last 15 years in a daily timeframe were downloaded. It can be observed that the asset begins near $61 in the oldest period of 2010 and oscillates around $118 as of October 16, 2025, the last selected data point. Upon observing the initial chart of the price movement over the selected time, a maximum near $250 is observed in 2014, while its lowest point was reached in 2020, below $50. This negative movement was widespread in the hospitality sector and in most of the different stock exchanges worldwide. It affected these companies for a long period due to major restrictions on global tourism, in addition to the high costs that had to be covered without being able to bring in their projected revenues.

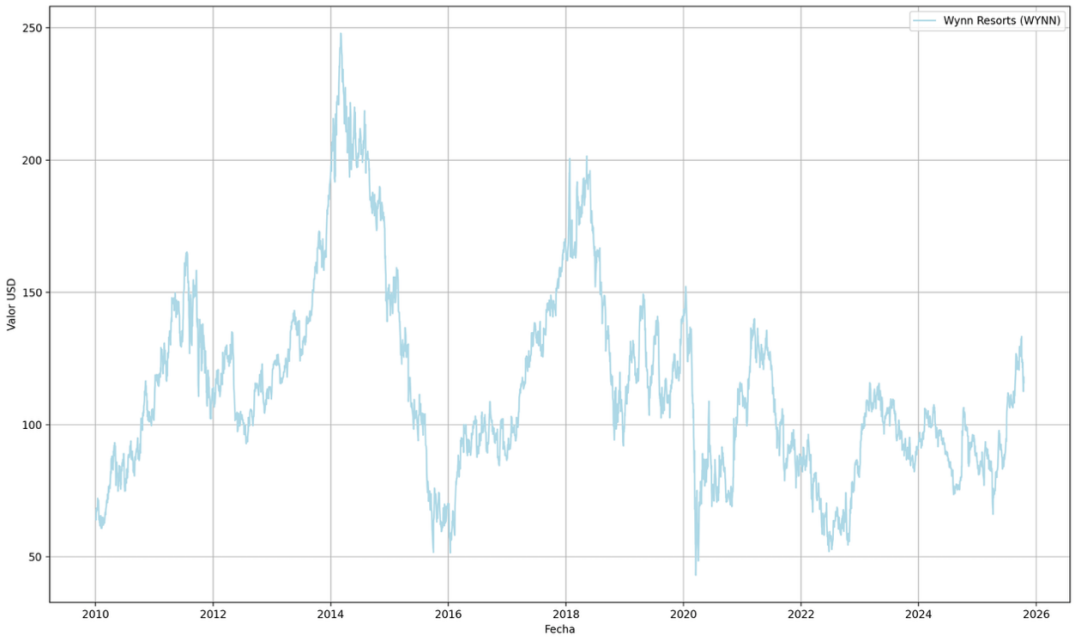



Chart 1. Price movement of Wynn Resorts (WYNN) from 2010 to 2025

## Strategy Overview

In this project, the aim was to design a systematic trading strategy for the prediction of buy, sell, or hold signals by analyzing the stock behavior of a specific asset. In this case, the strategy was applied to the asset Wynn Resorts (WYNN), chosen as an "atypical" asset that does not have the immense buy or sell volume but still maintains a large presence in the hotel sector. The models in this project have been trained by applying a set of technical analysis indicators to generate and classify signals into sell (0), hold (1), or buy (2), allowing them to identify complex or non-linear patterns in the asset's historical data that could be complicated to follow or identify with the eye of a common person.

For the analysis of the WYNN asset, two neural network architectures have been implemented in this project: the Multilayer Perceptron (MLP) and the Convolutional Neural Network (CNN). This was done with the main objective of comparing the performance of both architectures and determining which of the two methods is more efficient when working with the asset's historical data. On one hand, there is the MLP, which works by treating each day's input as a flat vector of features and can come to identify interactions between indicators (TensorFlow, n.d.).

And on the other hand, there is CNN, which seeks to test the hypothesis of the existence of useful information such as certain temporal patterns. It works by applying filters or "kernels" that move across short sequences of data, allowing it to identify certain patterns or motifs that could cause a movement in the asset's price (Brownlee, 2018).

While the applied Deep Learning approach can offer significant improvements when building a trading model, giving it a certain adaptability to make decisions automatically, it can still present limitations. In this sense, the main limitation is the risk derived from "data drift," as markets are not stationary. This could cause a model trained with data from 2010 to 2015 to fail and lose its predictive power when put to work with data from 2020 (Evidently AI, n.d.). That said, one of the main risks addressed in this project is the monitoring of this "data drift." And, being a trained computational model, there are also limitations regarding the risk of overfitting the

model, causing it to learn and adjust to the noise in the data instead of the signals. Therefore, the splitting of information into different datasets has been implemented with the objective of training (60%), testing (20%), and validating the model (20%) to ensure that the performance obtained in each dataset is a realistic approximation of how the model might operate with future data it has never seen before.

# Feature Engineering

# Data Pre-processing:

## Feature Analysis and Selection:

As part of the pre-processing, the insertion of technical analysis indicators was performed first, combining 3 types of indicators: momentum, volatility, and volume. Together, these 3 groups seek to obtain information to perform a detailed analysis and capture the behavior of the previously seen price dynamics.

The group of momentum indicators aims to capture the strength and direction of the price movement, allowing it to obtain the persistence of the observed movements, as well as the moment when its direction or trend will change. This can be useful for neural networks and is very common in the strategies of investors who use technical analysis. Among those considered for this project are RSI, MACD, ADX, and stochastic oscillators (all with more than one window).

On the other hand, the volume group is included seeking to measure the relationship between the flow of market transactions, considering the validity of a movement to be high if it occurs with high volumes. It also operates as a signal of confidence when the price moves in one direction and can warn of large investment outflows if high exit volumes occur. Some of these indicators are OBV (On Balance Volume), VWMAP deviation, and spike ratio. Finally, the volatility group aims to measure the uncertainty or the range of changes in the price, relative to its averages. This group helps to obtain information about the speed of the market, as well as the appropriate moments to execute trades, and it gives meaning to the other indicators, since abrupt

changes can occur during periods of high volatility, and vice versa in opposite periods. Some of those included are ATR, Bollinger Bands, and Donchian Width.

The goal of these combinations is for the model to gather enough information to interpret patterns and signals so that, when new data is introduced, it can identify shapes, paths, or strategies that ultimately generate a benefit — not only by improving the model's accuracy in predicting signals, but also by producing meaningful results that enhance the model's performance during the back testing of trading operations.

## Data Normalization:

From the moment data is downloaded, it is important to clean any missing values so that, when separating the information into features, targets, etc., this is done consistently and all variables remain complete and properly aligned. Afterwards, the dataset is divided into training, validation, and testing sets in order to prevent any information leakage between the training data and the other subsets.

For this experiment (and for most of the models previously tested, although some handle this internally), it is necessary to normalize the data since each feature or column involved can vary in both distribution and range, which can affect the model's performance and training stability.

Once the data was split, four different scalers were created, each applied only to the columns it is meant to adjust. The first one, a MinMaxScaler, was used for certain features such as RSI and Stochastic Oscillators, since their values range between 0 and 100. The second, a RobustScaler, was used for columns such as ATR and Donchian Width, which contain outliers during extreme or high-volatility periods of the asset. This scaler relies on the median and interquartile range to reduce the influence of those outliers. The third one, the most common, is the StandardScaler, applied to features that behave roughly like a normal distribution (mean near 0 and variance 1), such as MACD, ADX, and OBV. Although not all of them follow a perfectly normal distribution, their behavior is similar enough that treating them this way does not negatively affect model performance.

The last scaler, although also based on StandardScaler, was separated because it handles different information—specifically, the base columns obtained directly from the downloaded price data. This allows the model to stop relying on absolute prices and instead learn from their variations regardless of magnitude. In this way, correlations are maintained without being biased by scale differences, and volume values are adjusted to the same statistical range.

Finally, this separation was done to respect the inherent nature of each indicator without sacrificing real financial relationships. It also improves numerical stability during training and makes feature values easier to interpret. Performing all these transformations with a single scaler would result in information loss and distorted relationships, ultimately hindering both interpretability and model performance.

## Target Variable Definition

### Label Creation and Threshold logic for class assignment:

Once all features were generated, the creation of future returns was carried out using a *make_forward_return* function. The objective is to use these returns to subsequently label the data that will be used to train the models. This function shifts the closing price by a desired window size (This window value, as well as most parameters, can be modified from the *config.py* file.) and obtains the return for that window, which is referred to as the "horizon." It also trims the final data points for which the calculation cannot be performed, due to a lack of data for comparison. It was decided to define dynamic thresholds using percentiles, converting them into upper and lower limits depending on the desired quantile. In this way, we did not obtain beneficial results, as the classes were highly unbalanced and caused problems starting from the training stage, even after applying class balancing later. Therefore, we decided to define fixed return thresholds and control the limit of the change in return that would trigger a label change.

The change made by defining fixed thresholds made it easier to visualize the consequences of the model and, by analyzing the asset's behavior, to adjust these limits toward the returns that would make operations more profitable. This was done

while considering that, during back testing, short positions would be penalized or charged an additional percentage due to the borrow rate, so executing too many trades would lead to losses if their profits did not justify the opening and closing of positions, since commissions are charged both when opening and when closing a trade, based on the asset's price at each moment.

Gradual adjustments were made to observe the effect of changing the thresholds, which ultimately showed that reducing the number of short positions is the only way to make sense of the results seen in the graph, given the high cost of these operations. From the configuration file, these parameters can be modified to ensure that the labeling results align with the intended objective. In this case, if high thresholds are left (which are rarely exceeded) and the resulting labeling is fed into the model, it produces a high accuracy but large losses. This means the model is not actually performing well — it is simply "guessing right" because most of the training data corresponds to the hold class. This indicates it is not truly learning, and when exposed to new data, it would produce inconsistent results even if the accuracy appears high. Another way to assess the labeling quality is through metrics such as Precision, Recall, and F1 Score, which in this case confirm that the model is underperforming. These metrics could not be integrated directly into the model's objective due to a library conflict, but they were analyzed separately.

## Distribution of signals in training data:

When running the initial tests with dynamic thresholds based on percentiles, we observed a clear imbalance toward hold signals, representing around 60–70% of all labels within the percentile range >0.8 (Upper) and <0.2 (Lower). After performing several tests and adjusting the percentile values, we noticed that this only made the strategies more aggressive (a greater number of trades) in order to meet the desired number of signals. However, this did not yield realistic results or a correct interpretation of the data. In short-term movements, the model could generate more than five signals within the same small price swing that would not actually produce profit due to the limited magnitude of the move. Additionally, it often identified the same trend multiple times, with most signals entering too late, resulting in financial

losses once trading and closing costs were considered. The situation was even worse for short positions, since they incurred not only commission fees but also daily borrow rate costs.

After deciding to use fixed thresholds for labeling, a clear improvement in the signals was observed. Although the issue of generating multiple signals for a single movement persisted, we gained more precise control over the sensitivity of the model—deciding whether to capture stronger or milder movements (based solely on the training data chart). Since the upper and lower limits were independent, we were able to penalize short trades more heavily, requiring a much larger price movement to open them. This ensured that any potential profit would justify and exceed the daily borrow rate and commission costs, especially given that the asset traded with relatively low volume, meaning slower price reactions and fewer opportunities for short-term profit.

In this way, although the imbalance problem remained, being able to fine-tune each threshold independently allowed us to see how the labeling visually aligned with the model's behavior and adjust it according to the desired strategy—which, in this case, aimed to capture mainly upward movements, consistent with the fact that during the training period the asset experienced significant price increases exceeding $200 USD.

It is also important to mention that the time horizon used to calculate all returns in this exercise was five days, as this captures the information of a full trading week (under regular conditions, with only a few exceptions that are filtered out at the beginning of the process). This horizon was chosen because labeling based on shorter future return intervals would introduce excessive noise into the signals, leading to many false signals during volatile periods. Therefore, using a sustained return over at least one week proved to be the most convenient and consistent adjustment.

To visualize our initial test, we plotted the points where buy, sell, and hold signals appeared—green triangles representing buy signals, red triangles for sell signals, and blue dots for hold signals or moments when no positions should be opened. In this first example plot, we used return thresholds that might appear reasonable for many assets, setting the labeling limits to only ±2% in either direction.

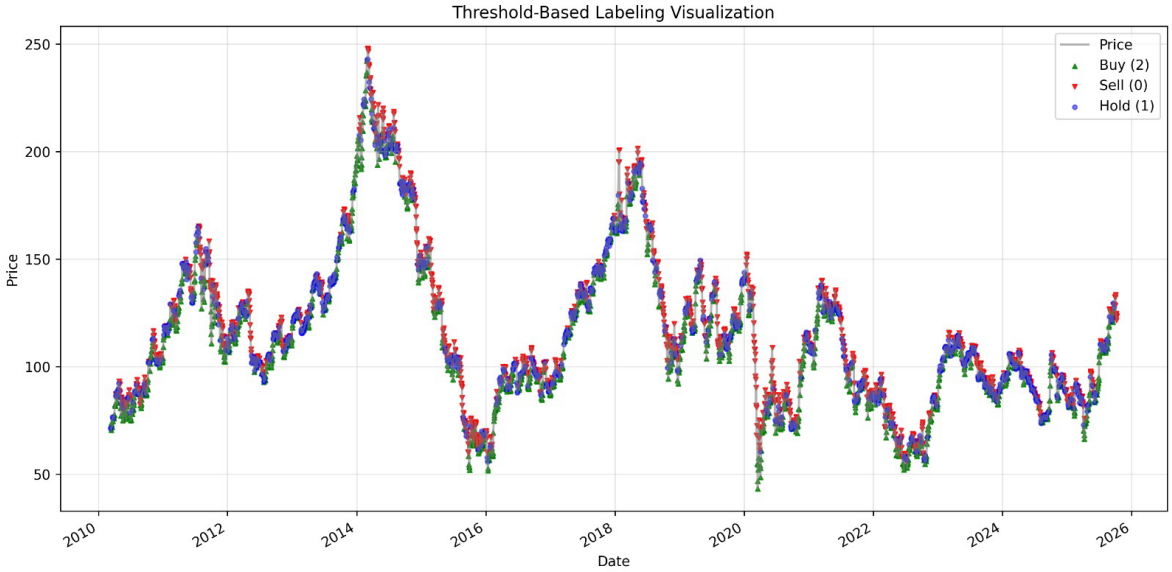



Chart 2. Labeling of signals with initial test parameters

As can be observed, we achieved a good proportion between buy and sell signals, exceeding those suggesting to hold the position. However, this configuration ended up generating significant losses in back testing due to the high costs associated with short positions. Additionally, the model failed to train effectively, reaching accuracy levels around 30%, meaning that its predictions were essentially random—or even worse than a random classifier.

As a second attempt, we aimed to adjust the thresholds more restrictively for sell signals (-0.1) and less restrictively for buy signals (0.00001), seeking to reduce the number of hold signals and ensure that the 5-day sustained return captured truly relevant movements.

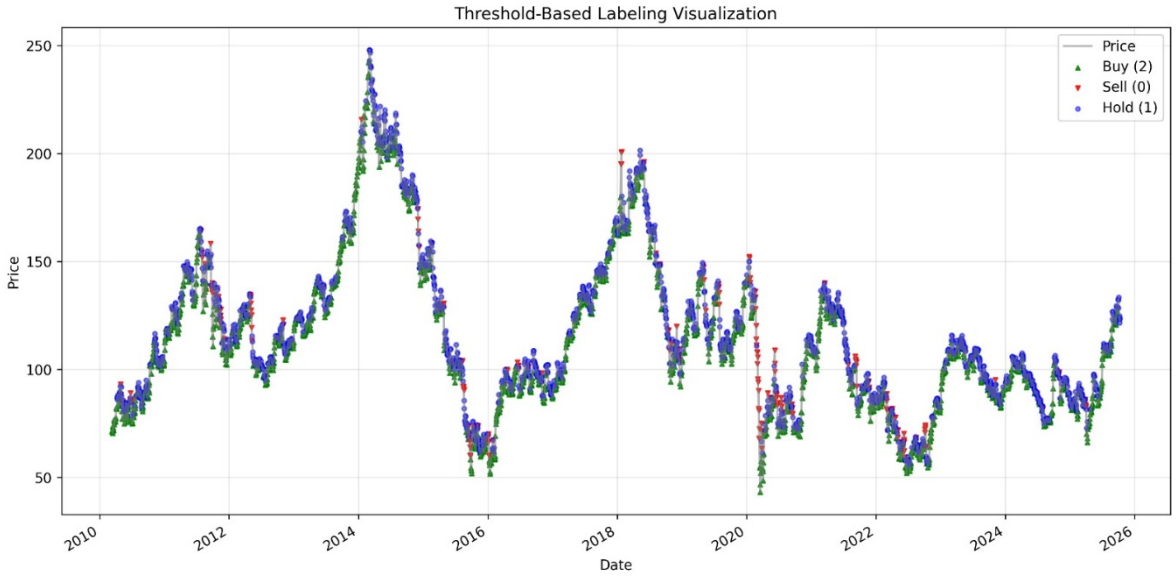



Chart 3. Labeling of signals with final parameters

In this way, a much higher proportion of buy signals was obtained, leaving only about 5% of sell signals. This led to better results in the model's accuracy, though not because the classification itself was functioning correctly, but rather because with hold signals representing 43% and buy signals 51%, the model had a higher probability of "guessing" one of these two classes, thereby reducing the likelihood of error.

We ultimately decided to keep these parameters (or make only very minor adjustments, as the proportions and performance remained nearly identical). After observing that these changes did not improve the model's loss function—and only increased accuracy—we concluded that the model was not truly learning effectively, and modifying this setup any further would not yield significant improvements.

# Model Architecture and Design

## MLP

The MLP model, a multi-layer perceptron neural network, was implemented with the goal of performing multiclass classification of trading signals (ultimately 0 = SHORT, 1 = HOLD, 2 = LONG). The implemented architecture followed a classical structure of dense and dropout layers, aiming to enable the model to learn effectively while controlling overfitting, so that it would remain useful when fed with new data—or, in this case, when predicting future trading signals to obtain an economic benefit.

The model received as input a vector of features (calculated as described in the feature engineering section) and was tested with different dense layer configurations, varying the number of units, the activation function (ReLU, Sigmoid), and the dropout rate (around 0.02–0.05) to ensure that neurons did not become interdependent and to prevent overfitting.

For the output layer, the SoftMax activation function was used, as it is responsible for performing multiclass classification—in this case, for the three trading signals. In addition, the defined hyperparameters included the optimizer (fixed as Adam) and the loss function (sparse_categorical_crossentropy), both chosen for their suitability and efficiency in handling multiclass classification with integer labels. The number of epochs was set to 40-50, after observing in multiple training sessions that this was more than sufficient to verify convergence, even though performance often began to degrade beyond 30 epochs.

This classifier does not take temporal sequences into account but rather focuses on the combination of indicators. For this reason, the model learns static patterns rather than dynamic ones. This is a major limitation, as it does not truly capture market volatility but instead seeks the best possible classification outcome, prioritizing accuracy. Ultimately, this undermines both the model's performance and the results of the back testing simulation.

For both models, class balancing was applied and taken into consideration, assigning weights in a more equitable way and penalizing errors in the minority

classes in order to reduce the dominance of the HOLD labels and achieve a more balanced outcome. Ultimately, however, it was observed that the balancing process was not effective—its functioning and results will be explained in detail later.

## CNN

For the CNN model, a lightweight one-dimensional (1D) convolutional network was implemented, unlike the 2D version covered in class. This decision was made because financial data does not have the same structure as image data, which the 2D network was designed to classify. In this case, we had to reshape the tabular financial data from simply (features) to a shape of (1, features), allowing the model to receive, train on, and interpret the data correctly. This way, we did not need to explicitly provide a temporal sequence to the model—the “1” mentioned here represents the timestep (since the data is daily and the features are calculated in the same way).

For the hyperparameters, we experimented with different numbers of filters (32, 64), kernel sizes (1–4), and activation functions (ReLU, Sigmoid), while keeping a fixed “causal” padding. This padding was chosen to preserve temporal causality, ensuring the model would not use future data for training but instead rely only on past and present values.

Regarding pooling, we used a size of 1 to maintain compatibility between the model structure and the data. We also tested different numbers of units in the intermediate dense layer (64, 128). For the output layer, we fixed the number of classes to 3 and used a SoftMax activation function, which produced a consistent and discrete result for multiclass classification, similar to the MLP model—allowing direct comparison and easier interpretation of the signals.

The training parameters were also fixed: we used the Adam optimizer and Sparse Categorical Cross entropy as the loss function. We experimented with batch sizes (32–64) and epochs (eventually set to around 50 across all models based on observed results), as well as the same class weights used in the MLP model. This

consistency ensured that the performance of both models could be compared fairly, without discrepancies caused by differences in weighting.

The purpose of this convolutional design was to capture interactions between indicators—such as relationships among RSI, MACD, OBV, and Donchian Width—rather than relying on just one or a few signals that could create saturated buy or sell triggers. The idea was to balance and uncover genuine relationships that would be useful for model training and prediction. However, the absence of temporal sequencing (a limitation that could be addressed by adding information directly to the data, outside the project's scope) prevented the model from fully capturing real patterns—at least with the data and setup we used. In another scenario, a temporal window could be implemented, i.e., a parameter representing several days, allowing the timesteps (currently 1) to evolve and reveal pattern dynamics. This would provide richer information about future behavior rather than relying solely on daily instantaneous values, which is especially important in trading where temporal sequences are crucial both for model performance and data interpretation.

The main metric optimized for both models was accuracy, although the final comparison was based on the lowest loss value, since both models showed poor loss results and did not function as reliable classifiers, even though they achieved accuracies above 0.45. Again, due to library conflicts, it was not possible to include Precision and Recall as main or additional evaluation metrics.

## Class weighting formula and application

Given the strong class imbalance observed at the beginning of this document, despite adjusting and modifying the labeling conditions, parameters, and horizons used for calculating future returns, we decided to implement class balancing using the scikit-learn library through its class weight computation function. This function provides a standard way to calculate class weights in problems where imbalance exists. We selected the "balanced" option for the computation based on the y columns, since the balancing is focused on the classes.

The formula used by this function to calculate the weights is:

$$w_i = \frac{N}{K * n_i}$$

Where $w_i$ is the weight of each class, $n_i$ is the number of samples in each class, $N$ is the total number of labels, and $K$ is the total number of classes.

De esta forma, el resultado que obtenemos es la magnitud de penalización de los errores para cada clase, por ejemplo, si una clase tiene tan solo el 5% (en este caso, para la clase de SHORT), cualquier error en esta clase, valdría 2 veces más que los demás, mientras que los errores de la clase con mayor distribución (la clase BUY o HOLD) tendría un valor inferior a 1. De esta forma se intenta compensar la falta de shorts, volviéndola más relevante para el modelo.

Finalmente, como una limitante para un caso real, vemos que el desbalance no logra ser compensado del todo, pues a pesar de los pesos hay una clara tendencia hacia los “hold”, esto se confirma con el buen desempeño del accuracy en algunas pruebas, llegando a más del 65%, pero que al momento de probar, nos dimos cuenta que estaba simplemente acertando por casualidad en las clases mayoritarias, y las señales de compra y venta no eran útiles, pues terminaban por perder dinero en la simulación del back test.

# MLFlow Experiment Tracking

## Experiment setup and logging structure

For the management and testing of the different architectures with multiple hyperparameters, a study was conducted which maintained a record of the experiments generated with each code iteration using mlflow. This was in order to monitor and have reproducibility in the project by saving each obtained version to then compare it with the ones already obtained to determine which of all the attempts resulted in the best model. In this sense, the project was configured in such a way that it enables the user to load an existing model by inserting the name and version

of the model they wish to execute or, alternatively, to execute the training of a new model again and take that one as the new starting point for the project's exercise.

## Parameters tracked for each model & Metrics recorded

For each code execution, a record was kept of the key parameters of the best model of each generation, that is, with each execution of the training module, three models of each type (CNN and MLP) are generated, which the code analyzes, saves the best version of [that] type, and then compares the best of each type to save the one that will be the best model of all. This works by logging two sets of information:

- Model Parameters “mlflow.log_params(params)”: Where values like “num_filters” and “kernel_size” for the CNN models, or “dense_blocks” and “dense_units” for the MLP models are included. In addition to these values, parameters were registered that served to help identify trends in the creation of each model. For example, the activation and the number of epochs that served to identify to what point the model could improve by changing these parameters.
- Performance Metrics “mlflow.log_metric(…)”: Which served to save the values of final_val_accuracy (validation accuracy) and final_val_loss (validation loss) to finally be able to compare the performance of each model.

## Model comparison table: MLP vs CNN vs results

After the experimentation period, a comparison between both models is generated to obtain the following table, which summarizes the results obtained in each model:

| Model Type | val_accuracy | val_loss | Best Model by Type |
|---|---|---|---|
| MLP1 | 0.4790 | 0.9462 | |
| MLP2 | 0.3783 | 1.0161 | MLP 1 |
| MLP3 | 0.3707 | 1.0556 | |
| CNN1 | 0.4497 | 0.9523 | |
| CNN2 | 0.4866 | 0.9575 | CNN2 |
| CNN3 | 0.4255 | 0.9671 | |

Table 1. Find best model experiment (example)

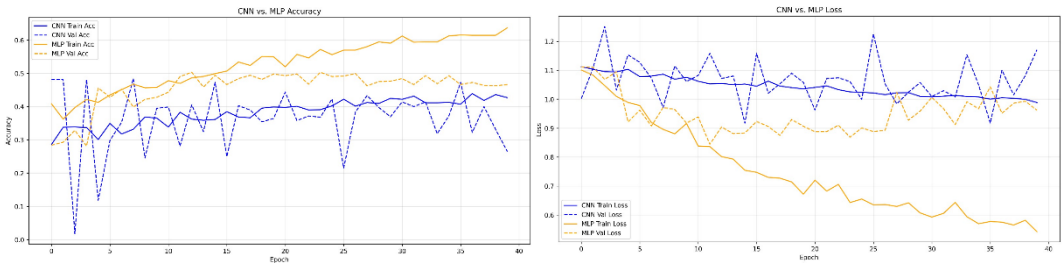



Chart 4 & 5. Evolution of accuracy and val loss during an experiment

This table and graph, for example, is the result of a training and the evolution of the accuracy and loss from which the best model would be MLP1, as it is the best model with balance in its accuracy and its loss. After selecting the model, its parameters are saved which, for this experiment would be: “{'dense_blocks': 4, 'dense_units': 128, 'activation': 'relu', 'dropout': 0.2, 'optimizer': 'adam', 'epochs': 40, 'batch_size': 32}”

## Selected model justification and performance on test set

| Best Model | |
|---|---|
| Name: | SystematicTradingModel |
| Model Type: | MLP_dense4_units128 |
| Version: | 3 |
| Accuracy : | 0.5376 |

Table 2. Best model obtained after experimental phase

Due to the way this project was built, the model selection logic in model_training.py is designed to prioritize the lowest validation loss (val_loss) over validation accuracy (val_accuracy). Now, to obtain this result, first the model training had to be executed over and over again until obtaining a decent model that could be functional given the

context of this project. After some attempts, the best model for this project was achieved, which was an MLP architecture (MLP_dense4_units128), registered as Version 3, which reached a final validation accuracy of 0.5376 (53.8%). Said model was saved and registered under the name SystematicTradingModel, serving as the production model for the back testing phase.

## Data Drift Monitoring

Regarding "data drift," an important assumption when applying these machine learning and "Deep learning" processes is that when training a model, it is assumed that the data used has a statistical distribution similar to that of the real data in the final validation stage or when releasing it to production. And as mentioned before, in reality, this does not happen in the markets. Based on this premise, given the non-stationarity of the markets, it is also expected that the volatility, variance, and mean of an asset's returns will change over time, rendering any model or algorithm with fixed parameters in its training inefficient (Evidently AI, n.d.). This phenomenon, as was already presented in the breakdown of this project's strategy, is one of the main risks when modeling an automated trading system, as it would only take something in its environment to change for it to lose its predictive power (like running a model with old parameters in current times, or running a model designed for a low-volatility market in a high-volatility market, and vice versa). Faced with this, a possible improvement would be to apply constant monitoring of "data drift" to prevent the model from failing suddenly, causing large losses.

To address this challenge, a statistical monitoring methodology was implemented in this project to quantify the "data drift" per "feature" that the model receives. One of the tests used is the "Kolmogorov-Smirnov Test" or "KS-test," which is an ideal non-parametric test for the project's continuous numerical features. It works by comparing the cumulative distribution functions of two datasets. Applied to this case, it compares the distribution of a feature in the "training" dataset with its distribution in the other two sets, "test" and "validation" (IBM, n.d.).

On the other hand, as a statistical analysis, there is also the "p-value," which indicates the probability that both samples come from the same distribution (IBM, n.d.). With this idea, a "p-value" below a significance threshold of 0.05 would indicate that there is sufficient statistical evidence to conclude that the distributions are different, and therefore the presence of "data drift" would be confirmed. Given that all the "features" of the project were numerical in nature, the KS-Test was the main methodology selected.

Said data drift detection was implemented in two phases. Initially, a static analysis was performed through the "drift_report.ipynb" notebook, where the p-values from the KS-test are calculated for each feature when comparing the complete "Test vs Train" and "Validation vs Train" periods. This provides a high-level overview of the feature stability between the large data blocks. In this case, the following table shows the features from this project that have shown significant drift using an Alpha of 0.05:

| | p_value_test | drift_detected_test | p_value_val | drift_detected_val |
|---|---|---|---|---|
| obv | 1.531e-316 | True | 0 | True |
| tr_norm | 1.556e-121 | True | 5.89e-15 | True |
| vol_sma_20 | 4.762e-66 | True | 2.266e-12 | True |
| Low | 1e-65 | True | 2.701e-172 | True |
| Close | 4.389e-65 | True | 1.547e-174 | True |
| Open | 1.71e-63 | True | 4.228e-175 | True |
| High | 4.429e-62 | True | 1.669e-178 | True |
| atr_28 | 1.642e-45 | True | 6.706e-187 | True |
| Volume | 4.614e-38 | True | 0.001579 | True |
| bb_width | 1.894e-30 | True | 1.337e-07 | True |
| atr_14 | 2.402e-23 | True | 3.611e-153 | True |
| rsi_28 | 1.826e-20 | True | 0.2193 | False |
| Adj Close | 1.087e-19 | True | 3.098e-57 | True |
| adx_28 | 1.322e-19 | True | 5.296e-17 | True |
| macd_signal_19_39 | 8.922e-15 | True | 1.435e-18 | True |
| macd_19_39 | 1.25e-14 | True | 4.418e-18 | True |
| macd_signal_12_26 | 4.785e-10 | True | 5.656e-13 | True |
| rsi_14 | 8.279e-10 | True | 0.3041 | False |
| macd_12_26 | 4.584e-08 | True | 5.551e-12 | True |
| roc_20 | 5.181e-08 | True | 0.04752 | True |
| mfi_14 | 1.204e-06 | True | 0.04892 | True |
| stoch_k_28 | 2.084e-06 | True | 0.03784 | True |
| vwap_dev | 7.152e-05 | True | 0.6206 | False |
| cmf_20 | 0.0007037 | True | 0.003458 | True |
| bb_percent_b | 0.001145 | True | 0.1475 | False |
| adx_14 | 0.002488 | True | 0.2134 | False |
| donchian_width | 0.009041 | True | 2.929e-55 | True |
| std_20 | 0.01636 | True | 9.654e-50 | True |
| williams_r_14 | 0.03037 | True | 0.1832 | False |
| stoch_k_14 | 0.03843 | True | 0.04021 | True |
| momentum_10 | 0.1503 | False | 1.544e-06 | True |
| vol_zscore_20 | 0.3252 | False | 0.2796 | False |
| vol_spike_ratio | 0.4517 | False | 0.6834 | False |

Table 3. Data drift per feature.

In the results of this graph, it can be appreciated that most of the features show significant data drift in both datasets. The interesting part of this result is seeing that

the features that showed more “stability” are those more related to momentum or small trends that end in short-term high-volume spikes (vol_zscore_20, vol_spike_ratio). These results with a high presence of data drift could represent a regime or trend change in the market that differentiates the training set from the test and validation sets.

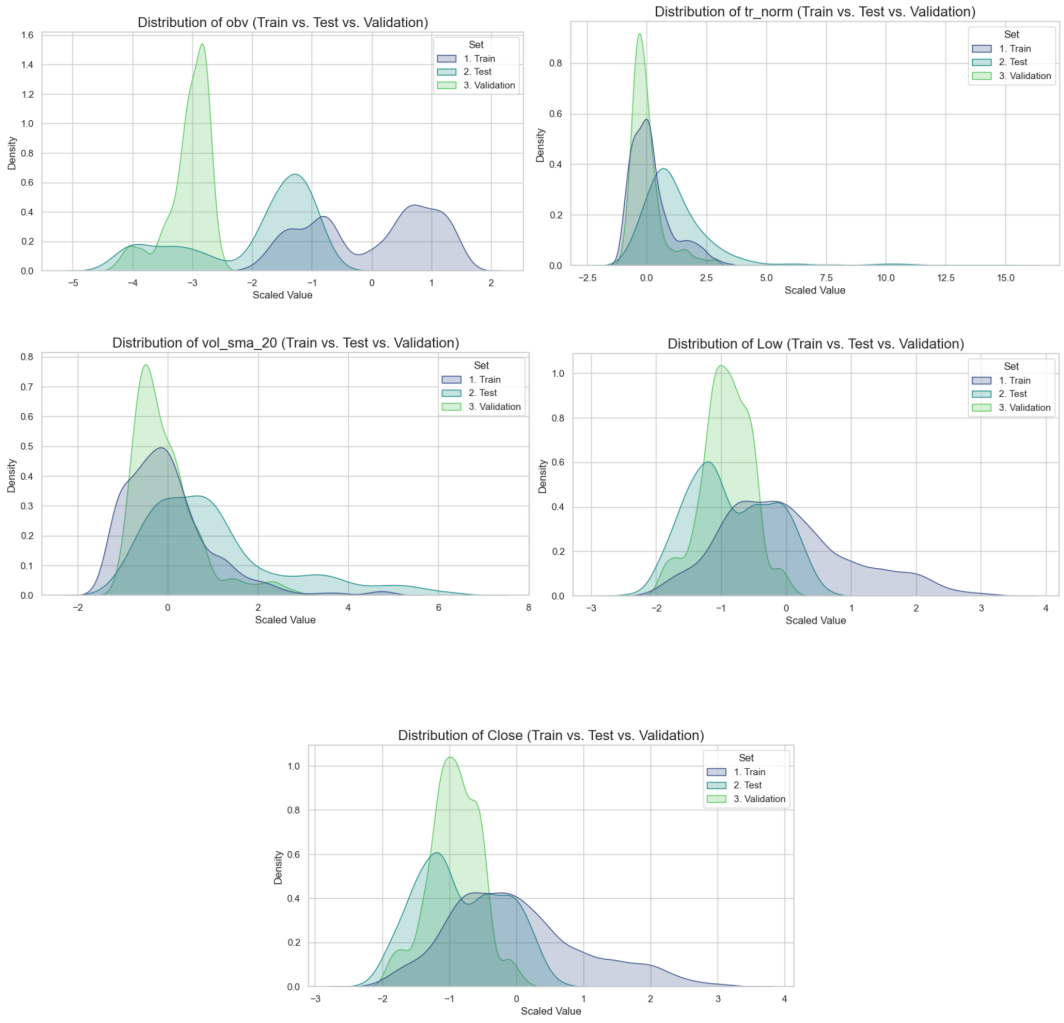
| | p_value_test | drift_detected_test | p_value_val | drift_detected_val |
|---|---|---|---|---|
| obv | 1.531e-316 | True | 0 | True |
| tr_norm | 1.556e-121 | True | 5.89e-15 | True |
| vol_sma_20 | 4.762e-66 | True | 2.266e-12 | True |
| Low | 1e-65 | True | 2.701e-172 | True |
| Close | 4.389e-65 | True | 1.547e-174 | True |

Table 4. Top 5 most drifted Features (Test vs Train).

| | p_value_test | drift_detected_test | p_value_val | drift_detected_val |
|---|---|---|---|---|
| obv | 1.531e-316 | True | 0 | True |
| tr_norm | 1.556e-121 | True | 5.89e-15 | True |
| vol_sma_20 | 4.762e-66 | True | 2.266e-12 | True |
| Low | 1e-65 | True | 2.701e-172 | True |
| Close | 4.389e-65 | True | 1.547e-174 | True |

Table 5. Top 5 most drifted Features (Validation vs Train).

By analyzing the features with the most data drift, it can be stated with more certainty that this “failure” may be due to a change in the market, as features like “obv” and “vol_sma_20” are indicators related to the change in volume in the asset's exchange, which makes sense considering the fall in WYNN's prices approximately between the 2019 and 2021 period. In addition to these two results, it is also found that within the top five are the volatility indicator “tr_norm” and the price data itself, “Low” and “Close,” which could suggest that this change in the market affected the natural way in which this asset was traded.




Charts 6-10. Top 5 most drifted Features (Distribution)

In addition to the results from the previous table, with these graphs, the afore mentioned can be confirmed, which is that in almost all cases the distribution of the train set is very different from the test and validation periods. This leaves the idea that both the data and the asset's behavior with which the model was trained are very different from the data that the following two periods received.

To simulate a more realistic production environment, the static methodology was extended to dynamic monitoring integrated directly into the backtest.py engine. Instead of comparing the complete sets, this method calculates drift in a sliding window. During the test and validation simulation, the backtest compares the distribution of the training set (baseline) against the last drift_window days of data (21 days) at each drift_step (every 21 days). The backtest.py script now records the

count of how many features exceed the drift threshold at each checkpoint, returning a new time series (drift_series). This drift count series is plotted alongside the equity curve on a secondary Y-axis, allowing for a direct visual correlation between model degradation (drift) and the strategy's financial performance. …

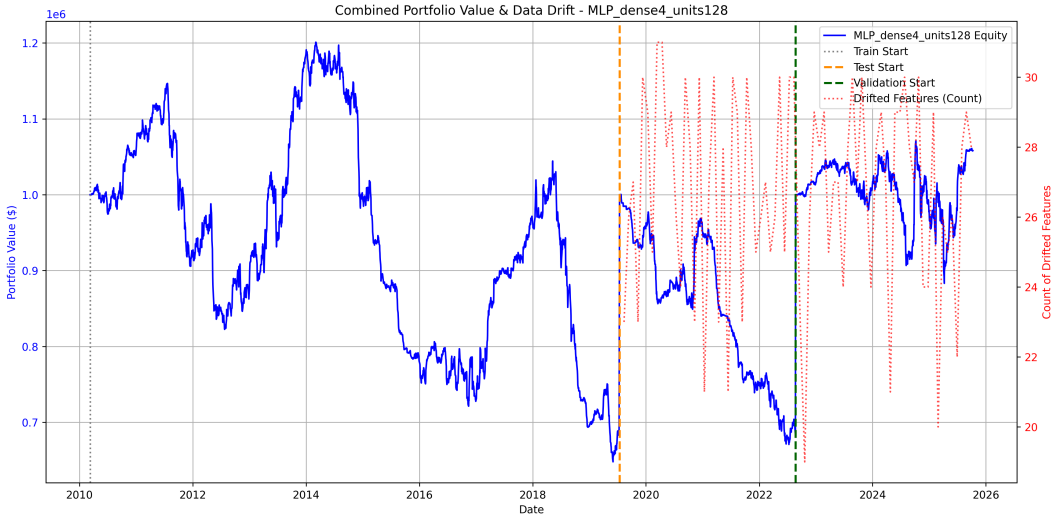



Chart 11. Features with drift over time.

Lastly, in this graph, the aim was to represent the evolution of the portfolio managed by the model alongside the count of features presenting drift, calculated as a dynamic window that updates every 21 days. The key takeaway from this visualization is seeing how it aligns with the previous idea, which is that from the moment the model begins to work with the test and validation periods, the data drift increases considerably, thus coinciding with the model's maximum drawdown period.

# Backtesting Methodology

## Signal generation from model predictions

Backtesting is what one can call the process of testing a trading strategy using historical data in order to study its viability and the strategy's own performance before risking real capital (Investopedia, 2023). In this project, a methodology is implemented that is designed to simulate the management of a portfolio with one stock, incorporating concepts such as transaction cost, a “borrow rate” which is the

payment made for keeping a short position open, which as its name says, is the price to pay for having the "borrowed" shares.

Once the best model is obtained (CNN or MLP, selected after the performance per model) it will generate a classification “prediction” to categorize by day the operation to be performed: 0 for sell, 1 for hold, and 2 for buy. These prediction signals (target) are received directly by the “backtesting.py” process, which translates them and activates the corresponding process by executing an order with the last closing price of the asset, for example, a “target == 2” would open a long position, a “target == 0” would open a short position, and a “target == 1” would generate an indication to keep positions open and not operate at that time

## Position sizing and management rules

One of the entry rules in this project that helps with the management of the portfolio's operations is “n-shares,” which establishes a fixed size of assets that can be traded per position, be it short or long. This rule is applied as the “position sizing” for this project and functions as a tool to control, in a certain way, the risk of the portfolio being traded (Investopedia, 2024).

Operation management, on the other hand, consists of fixed parameters for this exercise that act as restrictions to simulate a real scenario with existing costs and exits when operating the portfolio. These are:

- Transaction commissions

Which are incorporated as 0.125% of the operation's value and is applied to both the opening and closing of all positions, affecting the calculation of the real profit of each move made.

- Borrow rate (for short operations)

Applied to simulate sell operations more realistically with a rate of 0.25% annually. This rate is applied by calculating the cost, which is subtracted directly from the capital (cash) with each passing day, which likewise ends up affecting the real profitability or profit of each operation made.

- Portfolio management (stop loss and take profit)

Parameters that are defined arbitrarily and serve to control the closing of each operation as a fixed threshold (stop loss = 30% and take profit = 30%) on the closing price at which a position was opened. Within the backtesting, these thresholds are monitored for each position so that the moment the current price surpasses either of the two thresholds, the order to close and liquidate said position is issued.

## Assumptions and limitations of your backtest implementation

Despite the implementation of Deep Learning to generate a model to "predict market signals" based on multiple indicators, in the end, the backtesting process is a simulation based on assumptions that can differ significantly from real market behavior (QuantInsti, 2023). Therefore, even though it was built with a certain "robustness" by implementing commissions, borrow rate, and implementing parameters like stop loss, take profit, and n-shares that help in portfolio management, there will always be limitations given the assumptions of this part of the process.

Assumptions:

- Volume: It is assumed that it is always possible to buy or sell the number of shares specified by n-shares at that moment.
- Commission: It is assumed that this did not change in any of the 3 periods of this exercise: "train", "test", and "validation".

Limitations:

- Static parameters: This applies to both the n-shares and the limits set for the stop loss and take profit. The fact that these parameters are fixed could generate performance problems with changes in volatility or external events that could affect the market.
- Application in different markets or assets: This is due to the model, as it was trained and optimized exclusively with the historical data of WYNN. This would imply that the strategy is not universal, and if applied in other contexts, it is to be expected that the exercise would fail.

# Results and Performance Analysis

## Equity curve plots: train, test, validation periods

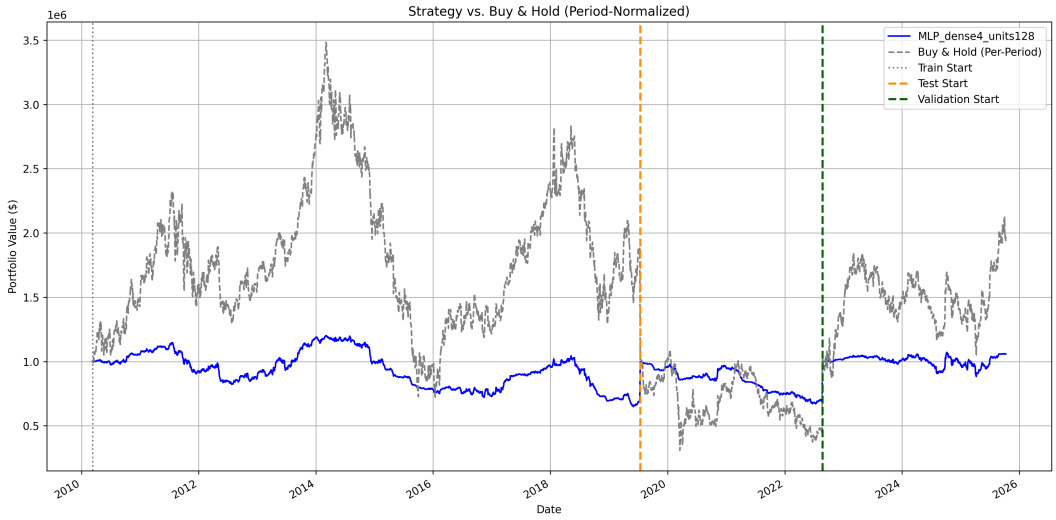



Table 12. Data drift per feature.

As a result of the best model obtained, the results of this graph were had, which shows the per-period comparison between a buy and hold portfolio and the portfolio managed by the model, both reset to one million dollars at the beginning of each period.

The first point to highlight is that in the three periods, the model achieved a certain "stability," showing it tried to follow the price movement trend of the WYNN asset. Something to stand out about this was the robustness the model showed when going through periods of high volatility and bearish trends. First, in the training set during the years 2014 and 2018; however, as this is the training period, this result will not be interpreted as definitive or correct.

However, as a second point, when comparing the performance on the test set, approximately from the end of 2019 to almost the beginning of 2023, the model achieved an interesting result; although it is true that it lost money, it must be considered that during this period WYNN suffered one of its worst price drops, which explains the fall of the buy and hold portfolio. Therefore, it is noteworthy that the model managed to resist the asset's downturn period, even managing to finish above

it. And finally, in the validation period, the model showed the same stability; in the more than two years of this period, it barely made a little money.

Although this result does not represent a good investment strategy given the relationship between the exercise time and the benefits obtained, in the end, we highlight the idea of the capability the model achieved to reduce losses in periods of high volatility, minimizing the erratic movement of a buy and hold portfolio that did not perform any operations in the period.

## Performance metrics & Trade statistics: total trades, win rate

| Performance Metrics | | | | | |
|---|---|---|---|---|---|
| Period | Final Cash (USD) | Sharpe Ratio | Calmar Ratio | Sortino Ratio | Max Drawdown |
| Train | $ 697,222.25 | -0.3200 | -0.0721 | -0.3255 | 0.4604 |
| Test | $ 685,846.31 | -1.4144 | -0.3571 | -1.5920 | 0.3294 |
| Validation | $1’058,135.93 | 0.2267 | 0.1340 | 0.2336 | 0.1753 |

Table 6. Performance metrics (Best model)

| Trade Statistics | | | | | |
|---|---|---|---|---|---|
| | Total Signals | | | Total Trades | Win Rate |
| Period | Long | Short | Hold | | |
| Train | 1033 | 162 | 1157 | 1195 | 70.37% |
| Test | 383 | 66 | 335 | 449 | 50.11% |
| Validation | 244 | 0 | 514 | 244 | 66.39% |

Table 7. trade Statistics (Best model)

Regarding the result of the performance metrics, perhaps one of the most interesting results was seeing the low correlation that can exist between the win rate and the model's performance, as even in the training period where the result of this metric is misleading when it is seen that despite having won 70% of the operations made, the model still lost more than $200 thousand dollars, a similar case that occurred in the test period where it is shown that the model managed to win close to 50% of the operations made but still also lost more than $200 thousand dollars. This would

indicate that the model won many small operations but the few it lost were large enough to counteract the gains.

## Model accuracy vs. strategy profitability analysis

Although the best model obtained had a validation accuracy of 53.76%. The trading statistics demonstrate that this metric is almost irrelevant for profitability. since profitability generally does not come from the frequency of hits, but from the magnitude of the gains versus the losses. In this sense, the 'Train' period (70% Win Rate, loss of more than $200 thousand dollars) is proof that a model can be "precise" and at the same time be a terrible trading strategy. Profitability was only found when the model, by chance or learning, stopped operating in one direction (Short) that was causing it massive losses.

## Conclusions

The strategy to develop a functional trading system was carried out according to the requested specifications, capable of transforming a traditional dataset in OHLCV format, with 15 years of daily frequency data, into a complete workflow that generates indicators, multiclass labeling (0 = Short, 1 = Hold, 2 = Buy), data normalization, training, and selection of deep learning models with the best performance according to selected metrics. The ultimate goal was to obtain useful results in a simulated backtesting environment that could potentially lead to profitability.

The results confirm that both the models and the proposed logic operate correctly; however, the simulated portfolio performance was poor when tested under “real” trading conditions. The model managed to learn only minimal structure and patterns from the data, showing apparently good accuracy during training but resulting in large financial losses, demonstrating that the models were unable to consistently capture the underlying information necessary to perform effectively in a live portfolio context.

The graphs show a clear difference between the portfolio’s value and its Buy & Hold benchmark, which implies much lower transaction costs since it only involves a long position. This highlights that the model and portfolio performance failed to capitalize on significant price movements in the asset. For instance, during strong upward trends, the portfolio’s returns lagged behind, as if the model could not recognize patterns signaling large-scale moves that could have generated profits.

In the portfolio value and data drift graph, we observe notable drops in performance during both the testing and validation phases, indicating that the relationships and feature distributions learned during the training period lost relevance when applied to subsequent unseen data.

Towards the end, the model shows a slight recovery during validation, roughly following the asset’s own upward trend in that same temporal segment. This suggests a modest classification and predictive ability, performing somewhat better

than a random classifier. In conclusion, the model demonstrates technical coherence and a consistent methodological process, but its economic viability remains far from being a profitable alternative for real-world trading use.

## Model selection and performance summary

The model with the best results was an optimal MLP (multi-layer perceptron) with the characteristics of 4 dense blocks of 128 neurons each, with a ReLU activation function, dropout of 0.2, 50 epochs, and a batch size of 32. This set of hyperparameters was the one that finally obtained the best balance and stability, prioritizing the accuracy of 53.76% on the test set, thus suggesting that it has a minimal understanding of the data structure, but that it fails to even double the performance of a random classifier.

This means that it does not achieve good performance and therefore, we cannot expect financial profitability within the backtesting. In the test and validation periods, a loss of money is shown when changes occur in the market, or in this case, of the analyzed and traded asset. The deterioration of the portfolio's performance increases as the data drift increases in the number of features; that is, as time passes, the initially trained distributions of the features cease to be relevant because they change after a certain time, resulting in high losses since the features do not provide sufficient or correct information to the model, which, in itself, was not a great classifier even when the distributions were the same.

We can assume that the model failed to capture consistent relationships or robust temporal patterns, both due to the limitation of the models, as well as the nature of the market's complexity, since only considering technical indicators, which as proven change their distributions as time progresses, means there is no financial profitability despite attempts to adjust parameters, the functions used for model activation, among other technical adjustments.

## Whether strategy profitable after transaction costs

The commission fees and borrowing rate represented a significant drawback for the trading operations, as the strategy did not generate sufficient returns to justify or

offset these costs in most trades. The profit margin achieved was too small, meaning that the system essentially provided a slower way of losing money rather than a mechanism for generating gains.

It can be said that the model managed risk more effectively, avoiding rapid exposure to market downturns (since it also incorporated short positions). However, it failed to correctly interpret market patterns to take advantage of short-term movements. Despite multiple adjustments and efforts, the overall strategy given its structure and the type of models implemented does not appear suitable for speculative trading or for generating consistent profits, primarily due to its inherent limitations.

## Recommended improvements or extensions

There are many ways to seek an increase in the model's effectiveness and robustness, mainly by improving the feature engineering, probably with variables that are no longer solely calculated based on the asset, but also on the market, such as regarding macroeconomic cycles, sentiment indicators, and liquidity indicators (something important to highlight is that this project assumes there will always be a sufficient quantity of shares or assets to carry out as many operations as we want at the precise moment and price we desire).

On the other hand, the trained models, at least in the way they were carried out, do not seem entirely useful, so it would be considerable to implement models of another type such as LSTM, RNN, DNN, or modify the CNN to one where temporality is considered, in order to capture better sequences that do not depend completely and exclusively on immediate features as suggested in the models section. This can also go hand in hand with regularization that seeks to avoid overfitting if dropout is not sufficient.

A potential improvement is the way the labeling was done, which may seem very basic, and could offer better results if carried out in a more conscious manner, for example, by adjusting a loss function that penalizes financial cost errors, not just basing it on the future returns sought, or also, a dynamic indicator that could be recalculated to observe if the model is decelerating or losing transactional volume to

receive different magnitudes in the model and know at which moments it is advisable or not to continue opening operations considering those same costs that diminish the final financial result.

This backtest simulation also assumes a mechanism that does not happen in real life, so it would be important before having a model we wish to launch into production or rely on for operations based on it, to consider more realistic market conditions and the risk and cost of an intermediary who will not always be willing to let us sell and buy with fixed rates, nor the volumes we desire.

With these improvements implemented, much better results can be obtained with better precision in signal prediction, performance, and metrics initially, but the implementation of a model of this type results in many limitations, at least with the instructions requested to be carried out.

## References:


- BM. (n.d.). Prueba de bondad de ajuste de Kolmogorov-Smirnov. IBM Knowledge Center. Recuperado el 26 de octubre de 2025, de https://www.ibm.com/docs/es/spss-statistics/29.0.0?topic=tests-kolmogorov-smirnov-goodness-fit-test
- Brownlee, J. (2018, December 10). How to develop convolutional neural network models for time series forecasting. Machine Learning Mastery. Retrieved October 26, 2025, from https://machinelearningmastery.com/how-to-develop-convolutional-neural-network-models-for-time-series-forecasting/
- Evidently AI. (n.d.). Understanding data drift and why it's so important. Evidently AI Blog. Retrieved October 26, 2025, from https://www.evidentlyai.com/ml-in-production/data-drift
- Investopedia. (2023, 11 de agosto). Backtesting: What it is, how it works, new model vs. old. Retrieved October 26, 2025, from https://www.investopedia.com/terms/b/backtesting.asp
- Investopedia. (2024, 21 de marzo). Position sizing: What it means, how it works, importance. Retrieved October 26, 2025, from https://www.investopedia.com/terms/p/positionsizing.asp
- QuantConnect. (s.f.). Walk forward optimization. Recuperado el 27 de octubre de 2025, de https://www.quantconnect.com/docs/v2/writing-algorithms/optimization/walk-forward-optimization
- QuantInsti. (2023, 12 de julio). Backtesting trading strategies: A complete guide. Retrieved October 26, 2025, from https://www.quantinsti.com/articles/backtesting-trading/
- TensorFlow. (n.d.). Time series forecasting. TensorFlow Core. Retrieved October 26, 2025, from https://www.tensorflow.org/tutorials/structured_data/time_series