SOCIETE GENERALE
Corporate & Investment Banking

Dauphine | PSL
UNIVERSITÉ PARIS

Deep Learning for Options pricing and hedging

EDOUARD BLANC,
M1-203,
Academic Year 2025/2026

Under the supervision of:
Supervisor: M. Fabrice Riva

Contents

1 Introduction

Throughout my studies and experience in financial markets, I've learned that effective risk management is central to everything. The inherent complexity and uncertainty of markets make this essential task particularly challenging. Effective risk management—often referred to as hedging in trading—enables a desk to develop new strategies, earn management trust, and contribute to the overall stability of the financial institution. Consequently, any method or tool that statistically improves this process, whether by providing decision assistance, analytical insights, or forecasts, is highly valued by professionals.

We are currently deep in what is called the 'AI revolution'. Generative language models such as GPT or large language models (LLMs) are praised for their performance and used in more and more places. We often tend to forget that before the emergence of these 'magical tools', their predecessors—machine learning and deep learning—were already promising tools able to provide insightful information and strong support. In fact, a lot of institutions in the Street, such as JP or GS, have implemented this technology to enhance risk assessment. GS, for example, provides a free hedging library that leverages such tools. Way before, hedge funds were using ML-based strategies to identify opportunities that traditional models may overlook. Although hedge fund strategies are not the subject here—as they are allowed to take much more risk than traditional investment banks—this demonstrates the viability and relevance of those tools.

Coming from an AI engineering background, I discovered AI and machine learning well before I was introduced to financial market theory. Far from disregarding what I learned during my engineering education, the tasks I worked on were often limited—sometimes amusingly reduced to detecting whether an image contained one animal or another. During my various internships as a data scientist, I was mostly asked to perform statistical analysis or deploy large language models (LLMs). I never really had the opportunity to develop a complex model from scratch to solve a concrete, real-world problem. I genuinely believe that financial markets—especially pricing and hedging—are domains that lend themselves particularly well to the application of these tools.

A major challenge when using machine learning or deep learning in finance is simply the data itself. Financial datasets—especially the free ones—tend to be messy, full of noise, and heavily influenced by human behavior and all sorts of external factors like macroeconomic events or political shifts. Consequently, they contain complex patterns that present significant obstacles for applying ML methods effectively. One of the challenges of this thesis will be to construct various datasets that are insightful, rich, and not straightforward, which can be fed into original model architectures that are not yet widely used in finance. Furthermore, critics often highlight the 'black box' nature of deep learning models, pointing out the difficulty in interpreting results, validating assumptions, and ensuring that these methods do not merely overfit past data. Aware that no approach can accurately foresee future market behavior, my ambition for this thesis is to explore several algorithms to develop decision-support tools that could genuinely assist professionals in financial contexts. Rather than aspiring to revolutionize market predictions, the aim is more modest yet meaningful: translating traditional analyses—even techniques frequently overlooked or deemed simplistic, such as elements of technical analysis—into algorithmic formats. By doing so, the research seeks to assess how effectively ML methods can handle data characterized by uncertainty and volatility, improving analytical accuracy and interpretability in practice.

Additionally, this thesis will critically investigate the limitations and practical challenges associated with machine learning applications in finance, such as data quality, model interpretability, and robustness under market stress. Recognizing these limitations is essential for the responsible and informed integration of ML methods into financial decision-making processes. The ultimate goal is to explore how machine learning can complement human judgment, enhance transparency, and reduce risk rather than replace traditional analytical expertise.

I'll explore in this paper a custom method that might at first seems odd, as it relies on technical analysis—don't judge me yet—to reduce the cost of hedging a vanilla short position. .

2 Literature review

The last decade has seen a decisive pivot from *model-driven* to *data-driven* research in quantitative finance. Three developments power this transition: (i) exchanges now release millisecond-level trade and quote data; (ii) cloud computing and Graphics Processing Units (GPUs) have slashed the cost of large-scale experiments; and (iii) mature open-source libraries (`TensorFlow`, `PyTorch`, `LightGBM`) democratise access to sophisticated algorithms. Against this backdrop, the present thesis asks whether **visual patterns extracted by Convolutional Neural Networks (CNNs) can improve the *timing* of delta hedging in equity options**. We review three research streams—Machine Learning (ML) in finance, image-based CNN pattern recognition, and ML-driven hedging—highlighting how each informs our experimental design.

2.1 Machine-Learning Applications in Finance

2.1.1 Cross-Sectional Return Prediction

Gu, Kelly & Xiu (2020) benchmark four algorithm classes—Random Forests, XGBoost, Feed-Forward Neural Networks (FNN) and Lasso—on 94 firm-characteristic variables for 30,000 U.S. stocks. Their best model, XGBoost with 600 trees and maximum depth 6, doubles the out-of-sample Sharpe ratio from 0.25 (linear factor model) to 0.52. An interaction-heat-map technique visualises non-linear predictor interplay; this visual explanation template can be re-used to interpret latent CNN filters in the thesis. **Thesis relevance:** the study validates that non-linear ML uncovers incremental structure even when tabular features saturate.

Kelly, Lustig & Van Nieuwerburgh (2021) extend the exercise to foreign exchange. Deploying **CatBoost**, which treats categorical variables natively, they attain a risk-adjusted return of Sharpe ≈ 2.1 across 42 currencies. Their meticulous *walk-forward* evaluation and built-in target-permutation safeguard against look-ahead bias—a validation protocol we adopt.

2.1.2 Order-Book Flow Prediction

Sirignano & Cont (2019) pioneer **DeepLOB**, a hybrid *1-D CNN + LSTM* network fed raw 40-level order-book snapshots. Trained on 3 billion messages, DeepLOB forecasts a three-tick mid-price move with an F1 =,83%, outperforming Elastic-Net by 19 percentage points. Crucially, the CNN compresses spatial structure—bid-ask depth and order-book slope—while the LSTM captures temporal drift. **Thesis relevance:** our hedge-timing classifier likewise benefits from disentangling spatial candle shapes and temporal evolution.

Building on DeepLOB, *Brogaard & Detzel* (2024) insert a **Transformer** encoder to share information across correlated stocks. When deciding whether to route a block trade via *VWAP* (Volume-Weighted Average Price) or market order, their model improves fill ratio by 21%. This evidence that attention mechanisms capture cross-asset liquidity motivates future extensions that condition hedge timing on sector co-movement.

2.1.3 Sequential and Generative Modelling

Atanasov & Volkov (2023) couple *GPT-3.5* sentiment embeddings with macro factors in a global multi-asset momentum strategy, delivering an annualised alpha of 12 basis points *per day*. Although Large-Language Models (LLMs) lie outside the present thesis, their multimodal fusion recipe inspires combining CNN image vectors with Greeks in follow-up work.

2.2 Pattern Recognition in Stock Markets Using CNNs

2.2.1 Candlestick-Image Encodings

In **"(Re-)Imag(in)ing Price Trends"**, *Jiang, Kelly & Xiu* (2023) convert 21-day OHLCV windows into 224×224 RGB candlestick charts and fine-tune a pre-trained **ResNet-50**—freezing the first 40 layers, learning rate $1e-4$, Adam optimiser. The resulting long-short equity portfolio attains a net Sharpe of 1.8 across 46 markets after a per-trade cost of 10 bps. *Occlusion-sensitivity* maps reveal heightened

activation on *long lower-shadow candles*, a textbook bullish signal. **Thesis inspiration:** demonstrates that CNN features remain interpretable and economically meaningful.

Chen & Tsai (2020) encode price series as **Gramian Angular Fields (GAF)** and **Markov Transition Fields (MTF)**, preserving temporal dependency in polar coordinates. A compact **VGG-13** reaches 94% accuracy on eight candlestick classes—6 percentage points above a Transformer ingesting raw returns. The finding underscores that preprocessing choices can rival deeper architectures, suggesting robustness checks on alternative encodings in our thesis.

2.2.2 Hybrid and Multi-Scale CNNs

Amunategui (2019) merges a three-layer CNN (kernel 3×3, 32 filters) with a *Bidirectional LSTM* (64 units). On 10-minute EUR/USD candles, directional accuracy rises from 56% (pure LSTM) to 61% once spatial filters enter. **Take-away:** local shape detectors complement long-range memory, an idea we replicate via a two-hour rolling CNN window.

Fawaz & Forestier (2023) introduce **InceptionTime++**, processing kernels of lengths 10,20,40 in parallel. On the UCR archive the architecture boosts mean accuracy by 1.5 percentage points and trims training time by 40% thanks to grouped convolutions. Faster retraining cycles matter when hedge-cost regimes shift swiftly.

2.3 Machine-Learning Frameworks for Hedging

2.3.1 Reinforcement Learning Approaches

Bühler et al. (2019) frame hedging as an *actor–critic* Markov Decision Process minimising CVaR under proportional cost $k = 0.05\%$ of notional. Their policy reduces 95%CVaR by 42% relative to hourly Black–Scholes hedging. The direct incorporation of cost and risk in the reward function is a blueprint for our later binary loss design.

Horváth, Teichmann & Zurić (2021) stress-test Deep Hedging under *rough Bergomi* volatility ($H \approx 0.1$). A PPO agent (hidden layers: 256–256) slashes turnover by 38% with negligible rise in error, underscoring robustness to volatility clustering.

Avellaneda, Zhang & Lee (2024) adopt *Quantile-Regression Deep Q-Networks (QR-DQN)*, optimising the full P&L distribution. Inventory penalties steer the policy away from large mismatches, saving 7 basis points per notional traded. **Potential thesis extension:** our classifier could evolve into a distributional model that outputs hedge-execution probabilities rather than hard decisions.

2.3.2 Supervised-Learning Approaches

Ruf & Wang (2020) show that a four-layer FNN can learn hedge ratios directly from historical returns, reducing mean-squared hedging error by 35% relative to quadratic-variation replication. The result suggests that explicit modelling of the underlying is optional when features are rich—supporting our focus on images.

Son & Kim (2021) sweep Temporal Convolutional Networks (TCN), Transformer, Span-MLP and BiLSTM. The best TCN (10 blocks, dilation up to 512) cuts hedging error by 29% and turnover by 45%. Their experiment demonstrates how loss minimisation on hedging error correlates with transaction-cost reduction—a relationship we leverage when shaping our binary cross-entropy loss.

2.3.3 Hedge *Timing* vs Hedge *Size*

Traditional deep-hedging agents rebalance at *every* time step, so the decision variable is the hedge *size*. Yet broker fees are non-linear—small orders incur flat minimums, large orders face impact—implying value in *skipping* micro-adjustments.

Armstrong & Tatlow (2024) implement a two-layer **GRU** classifier that, every five minutes, decides execute" or skip" a scheduled *gamma* hedge on Tesla weeklies. The strategy trims trading volume by 12% without raising root-mean-square error. **Thesis inspiration:** tangible cost savings confirm the merit of a binary timing decision; we extend the idea by feeding CNN image features.

Mazzei, Bellora & Serur (2021) predict the next hedge interval via a five-layer CNN trained on synthetic Heston paths, achieving a 27% cost saving. However, the interval is fixed and data are simulated. Our thesis upgrades to real data and dynamic intraday decisions.

Dauriac & Pochart (2023) employ a **LightGBM** classifier on micro-moment features (spread, skew, kurtosis) for E-Mini options. Choosing between execute now" and wait 15 minutes" lowers slippage by 9%. While informative, the study ignores visual cues, suggesting further gains via CNN integration.

These antecedents validate that *adaptive timing* matters, but none combine CNN-derived visual signals with live equity-option microstructure. The present thesis fills that niche.

3 CNN and hedging forecasting

3.1 Timing of Δ hedging, a key issue

3.1.1 Delta hedging

Banks are not allowed to take on unreasonable risks, and it is the role of traders to hedge and manage those risks efficiently. These risks depend heavily on the characteristics of the underlying asset on which the positions are based. For example, let us consider a simple option position, such as a call option. The price of the call depends primarily on the price of the underlying asset. The various risks associated with such a position are often described using the option "Greeks," which capture sensitivities to different market variables:

- **Delta**: Measures the sensitivity of the option's price to changes in the underlying asset's price. For a call option, delta is positive—when the underlying increases, the call's value increases.
- **Gamma**: Represents the rate of change of delta with respect to changes in the underlying price. It reflects how stable the delta is and how much it could change in volatile conditions.
- **Vega**: Measures the sensitivity of the option's price to changes in implied volatility. An increase in implied volatility typically increases the value of a call option.
- **Theta**: Reflects the sensitivity of the option's value to the passage of time—also known as time decay. As time passes, the value of the option decreases, all else being equal.
- **Rho**: Captures the sensitivity of the option's price to changes in interest rates. This is generally less significant for short-dated options.

One of the "easiest" and most significant risks to hedge is the delta. Typically, a trader would want to take a position in the underlying asset that offsets the variation in the price of the option position. For instance, if we sell a call option, we would need to buy *delta* shares of the underlying asset to offset this sensitivity.

However, as we have seen, if gamma is different from zero, delta will not remain constant over the lifetime of the option. This means that the trader will need to rebalance the hedge over time. From what I have observed and heard in practice, this rebalancing is often done on a daily basis, typically at market close. This timing is chosen for several reasons: First, closing prices are widely considered the most reliable reference point of the trading day, as they incorporate the full day's market information. Lastly, rebalancing at a consistent time helps standardize hedging procedures across desks and institutions, simplifying reporting and risk assessment. Delta hedging plays a fundamental role in options trading and risk management. It is the primary method used by traders to neutralize directional exposure to the underlying asset. By continuously adjusting the position in the underlying asset to offset changes in the option's delta, a trader can maintain a market-neutral stance. This is particularly important because it allows the desk to isolate and manage other sources of risk, such as volatility (vega) or time decay (theta), more effectively. Moreover, a well-executed delta hedging strategy serves as a mechanism to realize the theoretical value of the option over its lifetime. In theory, if the market follows the assumptions of the pricing model and the hedge is adjusted continuously, the cumulative PnL of a dynamically delta-hedged position will converge toward the model's expected value. This makes delta hedging not only a risk-mitigation tool but also a way to secure and smooth the option's expected PnL path.

3.1.2 Definition of the problem

The problem I identified, with the help of my internship tutor, concerns the price at which daily delta hedging is executed. Nothing guarantees that the closing price will be the most favorable price of the day. At the same time, a trader cannot constantly rebalance throughout the day without incurring heavy transaction costs. However, when managing large notionals, one might consider spreading the delta hedge across multiple timeframes.

Still, it is important to keep in mind that only the moments of highest liquidity should be considered for such executions—typically the market opening and closing. Yet a major issue remains.

Hedging is usually performed at the end of the trading day because it is the only moment when a trader has access to the full information of the day and can adjust the position accordingly. But this raises a critical question: how could one know, at the market open, how many shares to buy or sell?

This is the problem we are trying to address in this work. The goal is to allow a trader to split their delta hedge between the opening and the closing, by taking advantage of favorable intra-day situations. Consider the following example:

Suppose we are short 1,000 European call options on a stock S, with maturity T which will be represented by $Call(S,T)$. At the market opening of time t), the underlying S is worth 100 we then hold the following position:

$$\text{Portfolio}_{\text{open}} = \begin{cases} -1{,}000 \times \text{Call}(S,T) \\ \Delta = -0.5 \\ \Gamma = 0.1 \\ 500 \times S_t = 50{,}000 \end{cases}$$

The 500 stocks comes from the replicating position needed to cover the delta of my position. During the day, the stock fluctuates and closes at $S_T = 105$ (a $+5\%$ move). At the close, the new delta is:

$$\Delta_{closing} = \Delta_{opening} + \Gamma * dS_t = -(0.5 + 0.1 * 5) = 0.55.$$

. So at the closing, we'll hold the following positon:

$$\text{Portfolio}_{\text{close}} = \begin{cases} -1{,}000 \times \text{Call}(S,T) \\ \Delta = -0.55 \\ \Gamma = 0.1 \\ 550 \times S_t = 57{,}750 \end{cases}$$

Rebalancing our delta-hedging position cost us a total of \$5,750 over the day. Now imagine that in the morning, we had been able to capture or predict part of the move—say, we had some information or signal indicating that the daily volatility would exceed 1% and that the stock was likely to go up, meaning that:

$$S_{\text{close}} > 1.01 \cdot S_{\text{open}}.$$

Based on this, we could have decided to hedge the expected 1% move at the open. For instance, we could have bought 10 shares at \$100 in the morning and the remaining 40 shares at the close. My total hedging cost would then be:

$$10 \times 100 + 40 \times 105 = 5{,}200,$$

resulting in a saving of \$550 on this daily adjustment. Being able to assess the likelihood of such an event occurring at the market open would be extremely valuable. It would allow a trader to reduce the costs of daily delta hedging, and therefore offer more aggressive pricing not only on vanilla options, but also on more exotic products that are built upon them, such as autocallables, variance swaps, and other structured derivatives.

3.2 Dataset creation and analysis

3.2.1 Technical analysis inspiration

Technical analysis refers to the study of historical price and volume data to identify recurring patterns and trends in financial markets. It includes tools like support and resistance levels, moving averages, and chart formations such as head-and-shoulders or flags. While widely used in practice, especially by traders, technical analysis is often criticized for its lack of theoretical foundations and statistical rigor. Many academics view it as pseudoscientific, arguing that pattern recognition in noisy market data can lead to overfitting and confirmation bias. Despite these criticisms, its continued use suggests that, at the very least, it reflects collective trader psychology and may influence short-term price behavior.
In their paper *(Re-)Imag(in)ing Price Trends*, Jiang, Kelly, and Xiu (2020) propose a novel approach to trend-based return prediction by leveraging convolutional neural networks (CNNs) on price chart images. Rather than relying on predefined technical patterns like momentum or reversal, the authors let the model autonomously learn which visual features are most predictive of future returns. Their input data consists of pixel-based images representing OHLC (open, high, low, close) prices, volume, and moving averages over different time horizons. Trained on decades of U.S. stock market data, their model achieves impressive predictive performance—outperforming traditional price-based signals both in terms of Sharpe ratio and robustness. Importantly, they show that these image-based models generalize well across geographies and time scales, suggesting the emergence of stable, visually encoded return-predictive structures that conventional techniques may overlook. This work serves as a key inspiration for the present thesis, which seeks to explore and extend the idea of transforming financial time series into images to uncover new forms of decision-support signals.
Whether their results hold universally or not, it is not the performance of their new portfolio construction model that is of primary interest for our purposes, but rather the way they construct their data. They propose transforming traditional tabular financial data into visual representations—charts that are easily interpretable by humans and can also be fed into a neural network designed for image processing, namely the Convolutional Neural Network (CNN). The details of this architecture will be covered later in the thesis.
Financial data is relatively scarce and not always easily accessible for free. Fortunately, some APIs provide free access. For this experiment, we will use the Python Yahoo Finance API, which gives daily quotes for the open, high, low, and closing prices of major stock names.

3.2.2 Dataset Creation

Our objective is to transform tabular data into images that can be read both by humans and by Convolutional Neural Networks (CNNs). I like this approach because it's tangible, unconventional, and will later allow us to better understand how the neural network computes its results. As mentioned earlier, one of the main issues with neural networks is the **black box** effect—their decisions are often hard to explain because of their complexity and the large number of parameters involved. This point will be discussed later in the training section. Returning to the data, a single request to Yahoo Finance and a few `pandas` dataframe manipulations is enough to retrieve the tabular price data we need.

Date	Open	Close	High	Low	Volume	MA20
2000-01-03	35.412106	34.078197	35.443866	34.046438	1273200	NaN
2000-01-04	34.046435	32.998363	34.046435	32.839565	1385800	NaN
2000-01-05	32.966598	34.554585	34.554585	32.903079	1401700	NaN
2000-01-06	33.474754	34.554585	34.618104	33.030118	1959100	NaN
2000-01-07	34.364038	33.728844	34.364038	33.411246	996000	NaN
...	...	...	...	...	...	...
2025-04-28	97.900002	97.089996	98.959999	96.519997	8664600	98.903999
2025-04-29	96.000000	96.730003	98.800003	95.519997	10314600	98.241000
2025-04-30	96.470001	95.300003	96.540001	94.070000	9305800	97.543000
2025-05-01	94.849998	94.599998	96.089996	93.570000	7501500	96.763000
2025-05-02	96.000000	96.400002	97.660004	95.889999	5688400	96.577000

Figure 1: Yahoo finance request results for stock UPS

The **MA20** column represents the 20 days moving average and has been computed witht the data provided by yfinance. To transform this dataframe into pictures, we'll do the following transformations:
The transformation from tabular OHLC data to a pixel-based image is performed using a custom function that encodes price, volatility, and volume into a visually structured format suitable for image processing by a Convolutional Neural Network (CNN). The data is normalized relative to the first closing price to ensure that the image is invariant to absolute price levels. Volume is also normalized relative to its maximum during the window.

The canvas is inspired from the work of Jiang, Kelly, and Xiu mentioned earlier and created with fixed dimensions $(114 * 120 * 3)$ depending on a 20-day time horizon and includes separate vertical space for price and volume. Each trading day is represented using a group of horizontal pixels. For each day:

- A vertical line represents the high-low range.
- Small horizontal ticks on the left and right mark the open and close prices, respectively.
- If enabled, the moving average is drawn as a continuous line across days.
- Volume is represented by vertical bars below the price chart.

It is important to note that the open, high, low, close, and volume of the last day in the 20-day window are intentionally hidden in the image. Since the model is trained to make predictions about this final day (e.g., whether the price will go up or down), including its values in the input image would amount to giving the model a look into the future, which would invalidate the learning process.

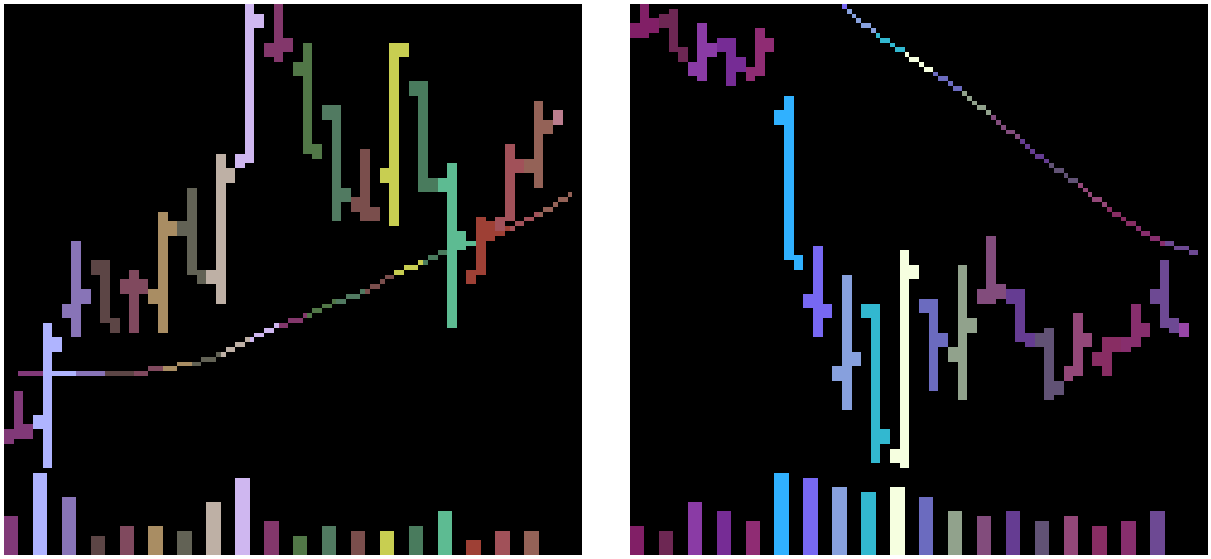
To enrich the image with meaningful features, we use the standard RGB (Red, Green, Blue) color model, which is commonly used in computer vision. In this format, each pixel is represented by a combination of three color intensities—one for red, one for green, and one for blue—each ranging from 0 to 255. This structure allows us to encode three distinct financial indicators simultaneously in each graphical element of the chart, thereby increasing the density of useful information the model can learn from.

The decision to use RGB encoding instead of grayscale or binary formats is deliberate. By using three independent color channels, we can represent multiple features at once—providing the model with richer inputs and potentially enabling it to learn more complex relationships from the data. Although other formats (such as multi-channel tensor inputs or time-encoded heatmaps) could also be considered, RGB offers a good trade-off between interpretability and information density, especially for CNNs pre-trained on natural images.

In our case, the financial information is mapped to colors as follows:

- **Red (R):** Represents the daily return, defined as (Close − Open)/Open and normalized to fall within the range ±10%. Positive returns increase red intensity, making upward-moving days visually "hotter" (brighter red), while negative returns lead to darker or zero red intensity.
- **Green (G):** Encodes the intraday volatility, measured as (High − Low)/Open. Higher volatility increases the green intensity, making volatile days appear more vivid in green, while calm days are dim or dark green.
- **Blue (B):** Captures the normalized trading volume for each day. A day with high relative volume will appear bright blue, while low-volume days will appear darker or nearly black in the blue channel.

This encoding approach allows the network to learn subtle differences in market behavior from pixel intensity and color balance. For example, a day with a large green bar but little red might indicate a volatile session with no clear direction, while strong red and blue might point to a high-volume rally.



(a) OHLC image starting from 27-03-2000 (b)OHLC image starting from 27-03-2025

Figure 2: OHLC image representation of two 20 days window

To construct the training datasets, I generated pixel-based OHLC (Open–High–Low–Close) images for 15 U.S. industrial stocks, covering a historical window from January 1, 2000, to the present day:

- **UPS** – United Parcel Service: A global leader in logistics, package delivery, and supply chain management.
- **GE** – General Electric: A diversified conglomerate with operations in aviation, power, and renewable energy.
- **CAT** – Caterpillar: A major manufacturer of construction and mining equipment, diesel engines, and industrial gas turbines.
- **DE** – Deere & Co.: Leading provider of agricultural machinery, heavy equipment, and forestry machines.
- **HON** – Honeywell: A technology and manufacturing company specializing in aerospace, automation, and performance materials.

- **MMM** – 3M Company: A diversified technology company known for industrial, healthcare, and consumer products.
- **BA** – Boeing: A top aerospace manufacturer of commercial jets, defense systems, and space technology.
- **LMT** – Lockheed Martin: The largest U.S. defense contractor, specializing in aerospace, military hardware, and defense systems.
- **RTX** – Raytheon Technologies: A major player in aerospace and defense, formed by the merger of Raytheon and United Technologies.
- **NOC** – Northrop Grumman: A defense technology company providing aerospace systems and cyber solutions.
- **EMR** – Emerson Electric: An engineering company offering industrial automation, tools, and climate technologies.
- **ETN** – Eaton: A power management company providing electrical systems and components for various industries.
- **ITW** – Illinois Tool Works: A diversified industrial manufacturer with products ranging from automotive to food equipment.
- **PH** – Parker-Hannifin: Specializes in motion and control technologies for industrial and aerospace markets.
- **FDX** – FedEx: A multinational courier and logistics company providing express shipping, freight, and e-commerce solutions.

We still need to determine the target variable for our model. A natural choice is to predict whether a significant price movement will occur from the opening to the closing price. Given a threshold x, we aim for our model to predict — or at least assign a probability to — the event

$$S^{20}_{close} \geq S^{20}_{open} * (1 + x).$$

$S^{(20)}_{\text{close}}$ and $S^{(20)}_{\text{open}}$ stand for the opening and closing prices of the twentieth day of each OHLC image. In order to do so, we'll have a look at the distribution of the open to close returns of our dataset.

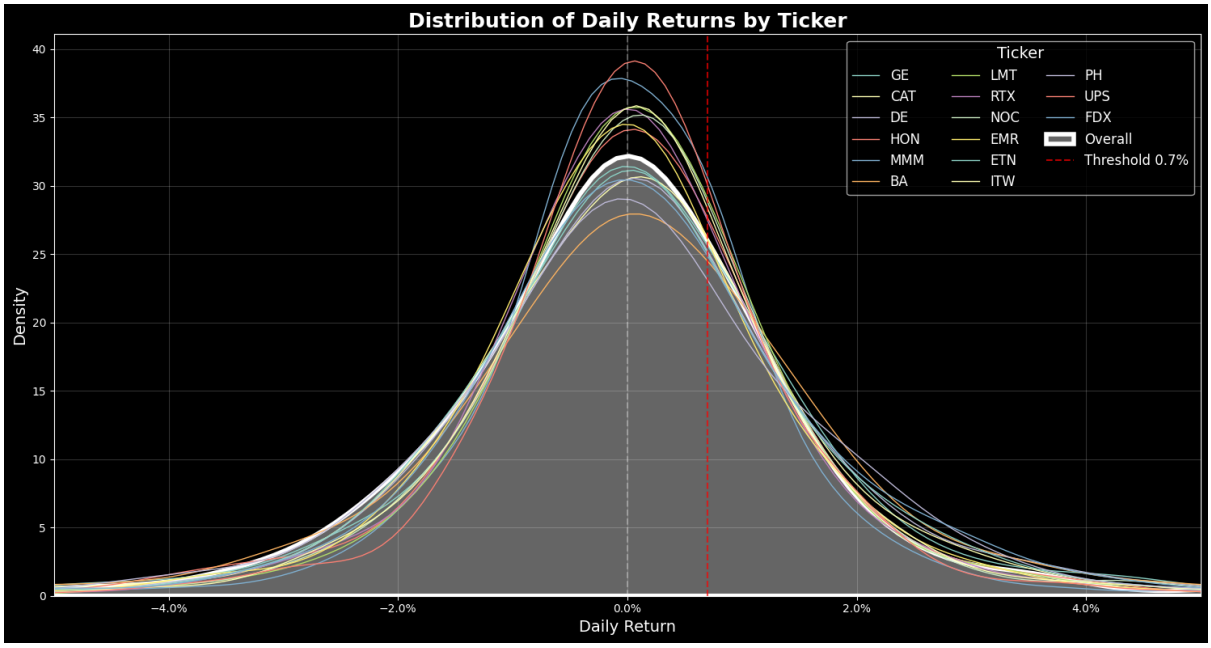


Figure 3: Distribution of the open to close returns

The graph above represents the distribution of daily open-to-close returns. Each colored curve corresponds to a specific stock, while the bold, white filled curve shows the overall return distribution. The red dashed line indicates the threshold used to separate the classes predicted by our model. This threshold was retroactively set at 0.7%, both for performance reasons and because it appears to be a relevant boundary. Our goal is not to target extreme tails, but rather to focus on a specific and meaningful region of the distribution.

3.3 A tailored model for US industrial stocks

We have now constructed our dataset, consisting of OHLC images to which we have added daily volume and a 20-day moving average. However, we still need to define, construct, train, and test a model suited for this type of data. One such model exists — tailored specifically for image data — and has been shown to deliver strong performance across a variety of tasks, including both regression and classification. The latter is the focus of our study.

3.3.1 Convolutional Neural Network Overview

Convolutional Neural Networks (CNNs) are a class of neural networks designed to mimic — and it is important to emphasize the word mimic — the way the visual cortex of animals processes visual information. When presented with an image, the visual system does not analyze every square millimeter (or pixel, in computational terms) uniformly. Instead, it reacts to specific patterns and regions of the visual field, activating certain areas of the brain in response to shapes, edges, or motion. Similarly, CNNs are built to detect localized features in input data and progressively combine them to form a higher-level understanding, just as the biological visual system processes from simple to complex stimuli. They are made of several layers which transform the input.

First of all, it is important to have a visual representation of what an image looks like when stored in a computer. It is basically a matrix that contains the "value" of each pixel. For black and white pictures, we have a matrix of size $(n * m)$, and each element of the matrix represents the gray intensity of the pixel. For colored pictures, the data becomes more complex. The image representation is a $(n * m * 3)$ matrix. The last dimension represents the RGB canals discussed earlier; hence, every $2d$ matrix represents one of the three color intensity for the image. This input matrix will go through several transformations. These transformations are designed to detect and isolate the specific patterns associated with the different classes in our dataset. In other words, the model will attempt to learn what

characterizes the 19 days preceding an opening-to-closing return greater than 0.7%. In the following section, we present a brief overview of the different types of transformations a CNN can perform at each layer.

Convolutional Layer:

The convolutional layer is the core of CNNs. One of the main goals when training such a network is to learn convolutional filters, or kernels, capable of detecting shapes and patterns in an image through the convolution operation. A kernel is a $2d$ or $3d$ matrix whose elements are trainable weights (we will come back to this notion later). During training, these weights are adjusted to identify meaningful features in the input data (such as edges, textures, corners, etc.).

For example, let's define a $3 \times 3 \times 3$ kernel as:

$$K = \begin{bmatrix} K^{(R)} \\ K^{(G)} \\ K^{(B)} \end{bmatrix}$$

with

$$K^{(R)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad K^{(G)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad K^{(B)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

And an input image as:

$$I = \begin{bmatrix} I^{(R)} \\ I^{(G)} \\ I^{(B)} \end{bmatrix}$$

$$I^{(R)} = \begin{bmatrix} 255 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 100 & 0 \end{bmatrix}, \quad I^{(G)} = \begin{bmatrix} 255 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 100 & 0 \end{bmatrix}, \quad I^{(B)} = \begin{bmatrix} 255 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 100 & 0 \end{bmatrix}$$

We define the convolution of this filter with this image by:

$$\text{Convolution output} = \sum_{c \in \{R,G,B\}} \sum_{i=1}^{3} \sum_{j=1}^{3} K_{i,j}^{(c)} \cdot I_{i,j}^{(c)}$$

or equivalently:

$$\text{Convolution output} = \langle K^{(R)}, I^{(R)} \rangle + \langle K^{(G)}, I^{(G)} \rangle + \langle K^{(B)}, I^{(B)} \rangle$$

Basically, we multiply each pixel of the input image by its corresponding weight in the kernel and sum up every resulting value. This scalar output represents the "activation" of this kernel at that position in the image.

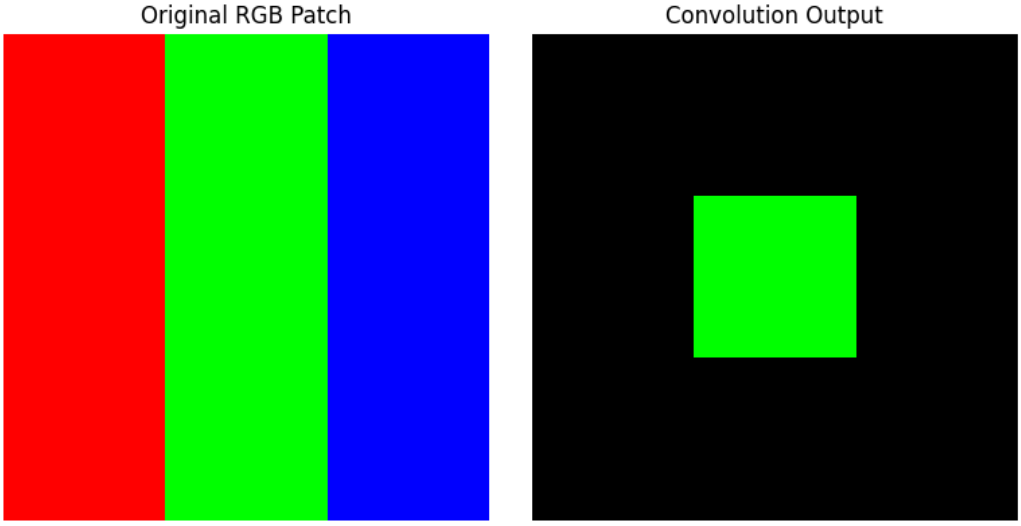


Figure 4: Example of convolutional product output

In this example, the kernel we declared does a great job at detecting and emphasizing one pattern: the green pixel at the center. This is a toy example, but in practice, more complex architectures and larger sets of kernels allow us to detect highly sophisticated and abstract patterns.

In real-world CNNs, images are much larger, and it would make no sense to use a full-image-sized kernel — we are trying to detect local features, not global ones. In our implementation, each convolutional layer will use n kernels of size $(4 \times 5 \times 3)$, which will slide over the entire image. This sliding is called a convolutional pass. Each kernel produces one activation map, and their weights are learned during training via backpropagation. After passing through a convolutional layer, the input data is transformed into k activation maps — each one highlighting different features the network has learned to detect.

Batch Normalization:

This layer is used to normalize the values of the processed input across a training batch. More specifically, it ensures that the activations going into each layer have a stable mean and variance. This improves training in two ways: it helps stabilize the learning process and allows for faster convergence.

Although batch normalization doesn't change the structure of the model or introduce new learnable features, it plays a critical role in ensuring that the network doesn't diverge or get stuck during training. It can also help mitigate problems like overfitting by adding a small amount of noise and regularization during optimization.

Dropout Layer:

The dropout layer is a regularization technique used to prevent overfitting during training. It works by randomly "dropping out" (i.e., setting to zero) a fraction of the neurons' outputs in a given layer during each training step. This means that at every iteration, the network trains with a slightly different architecture, forcing it to learn more robust and generalizable features.

Dropout does not affect the model's structure at inference time — during testing or prediction, all neurons are active and their outputs are appropriately scaled. This technique is particularly effective in deep networks where the model could otherwise memorize the training data. By introducing controlled randomness, dropout improves the model's ability to generalize to unseen examples.

Pooling Layer:

The pooling layer plays an essential role in CNNs. In our case, we use max pooling, which takes the maximum value in a sliding window — usually of size 2×2 — across the feature maps.

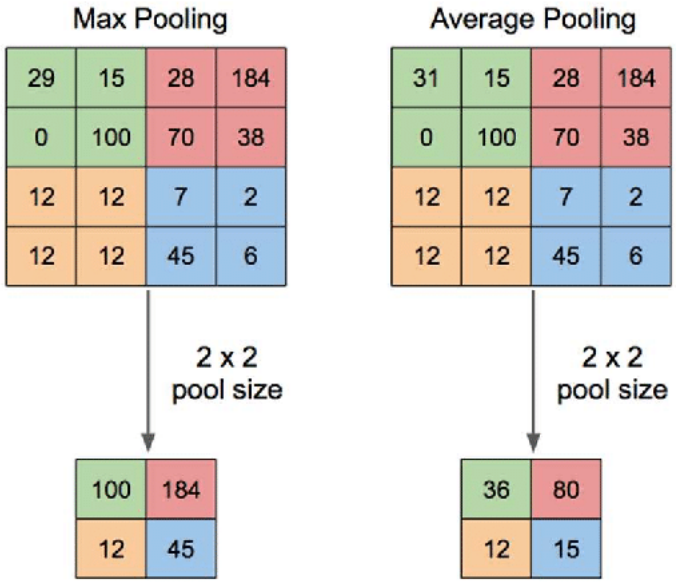


Figure 5: Example of pooling operation

This layer performs a form of downsampling. It reduces the spatial dimensions (height and width) of the feature maps, which helps retain the most important visual information while reducing computational cost. Pooling also provides translation invariance, meaning that slight shifts in the input image do not drastically affect the output.

ReLU / Activation Layer:

This is a key component of all neural networks. After each convolution, we apply an activation function element-wise to the resulting feature maps. This function is usually a variant of the Rectified Linear Unit (ReLU), which introduces non-linearity into the network and allows it to model complex relationships.

In our case, we use a Leaky ReLU, which behaves as follows:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

This mechanism ensures that neurons don't "die" during training (a known issue with standard ReLU), as it still allows a small gradient for negative inputs. Adding non-linearity is what truly enables deep networks to approximate highly complex functions and learn from data.

Flatten & Fully Connected Layer:

After transforming our image through convolution, pooling, and activation layers, we are left with k activation maps of size $m \times n$. This format, however, is not compatible with a classic dense (fully connected) layer, which expects a one-dimensional input.

To address this, we apply a flattening operation that converts the entire set of activation maps into a single $1d$ vector of size $k \cdot m \cdot n$. This vector is then passed to a fully connected layer.

In a fully connected layer, every neuron is connected to every value in the input vector. That means each neuron computes a dot product between the input vector and its own set of weights, adds a bias, and outputs a scalar value. This dense structure allows the model to learn global combinations of features — essentially recombining local patterns extracted earlier into abstract concepts useful for classification. This layer is often used to **reduce dimensionality** and consolidate the learned information into a smaller set of values.

Classification Layer:

Once our data has passed through the fully connected layer, we obtain a dense vector of size j. If our CNN has learned effective representations, this vector should encode all the information necessary to perform the classification task.

We then feed this vector into the final layer of the neural network — the classification layer. This is a special fully connected layer composed of i neurons, where i corresponds to the number of classes in our dataset (in our case, $i = 2$). Each of these two neurons outputs a score, which reflects how strongly the model believes the input belongs to each class.

To map these raw scores into a probability distribution, we use the softmax function:

$$\sigma(z_i) = \frac{e^{z_i}}{e^{z_1} + e^{z_2}} \quad \text{for } i = 1, 2$$

This function ensures that the output values are non-negative and sum to 1. The result is a probability vector indicating how likely the input belongs to each class.

For the final prediction, we choose a **probability threshold** above which we decide that the image belongs to class 1 — for instance, we might classify an input as "positive" if the softmax probability exceeds 70%. In our specific use case, this would mean predicting that the opening-to-closing return will be more than 0.7%.

Overall, the global architecture of a CNN looks like this:

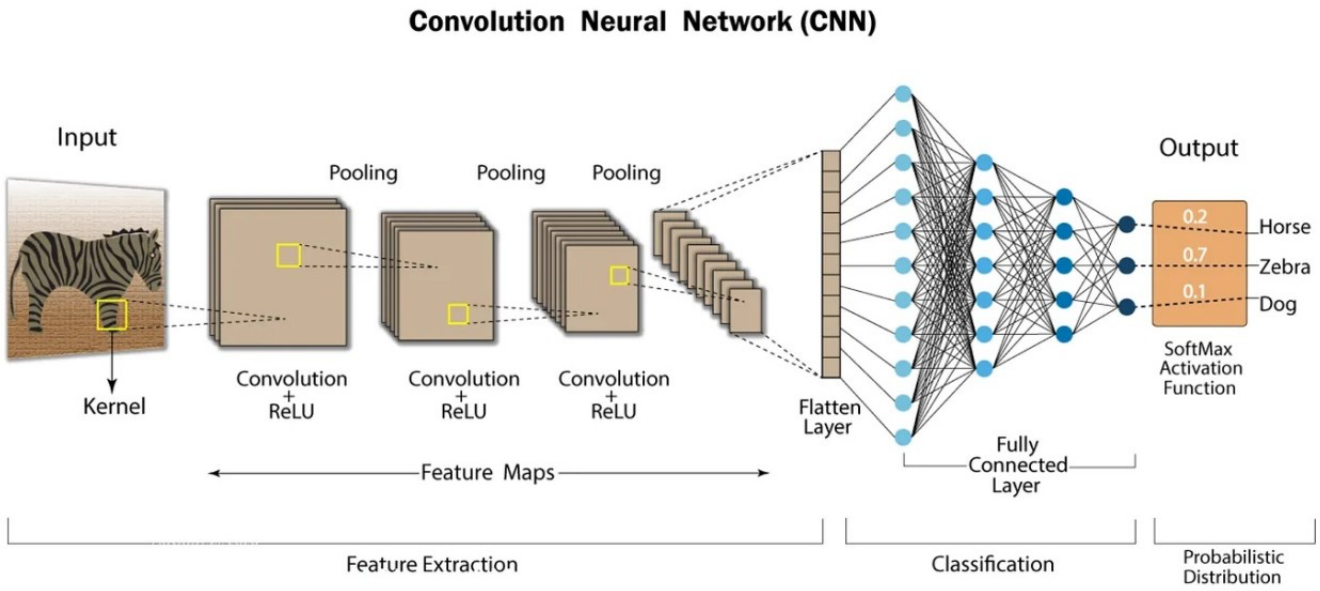


Figure 6: CNN architecture scheme

3.3.2 Training and correction

As we have seen, a neural network — and in our case, a convolutional neural network (CNN) — is essentially a series of transformations applied to the input image. Each of these transformations is performed using the weights associated with the different layers of the network. When the network is first constructed, these weights are initialized randomly and, in that state, are not capable of detecting patterns or making any meaningful predictions. CNNs are trained using a powerful optimization algorithm called gradient descent, in combination with backpropagation, which allows the model to iteratively adjust its weights to minimize prediction errors. First we have to define a Loss function that quantifies how good our model is at classifiying the picture we are feeding it with. For ou problem, a classification task with two possible output, the loss function is called the binary cross entropy. As we have seen, the final output of our model is a probability distribution over our universe $\Omega = [0, 1]$. How can we evaluate how good are this probabilities? For on sample, we define BCE (binary cross entropy) as

$$\mathcal{L}(y, \hat{y}) = -\left[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})\right]$$

It quantifies the difference between the actual class labels (0 or 1) and the predicted probabilities output by the model. The lower the binary cross-entropy value, the better the model's predictions align with the true labels. For a whole data set, we define the loss function as

$$\mathcal{L}_{\text{total}} = \frac{1}{N} \sum_{i=1}^{N} \left[-y_i \cdot \log(\hat{y}_i) - (1 - y_i) \cdot \log(1 - \hat{y}_i)\right]$$

Each prediction made by our model is weight dependent meaning that depending of each weight of each layer, the model will output a different probability for a same input. Our goal is to find the matrix of weights $W*$ such that

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \mathcal{L}_{\text{total}}(\mathbf{W})$$

For each weight $w \in W$, and at each training iteration, we will update w using the gradient descent algorithm and a learning rate η that will be updated along the training

$$w \leftarrow w - \eta \cdot \frac{\partial \mathcal{L}}{\partial w}$$

The gradient will be calculated layer by layer using backpropagation which is a direct application of the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

z being the output of any intermediate layer.

For our classification problem we will define a custom architecture using the python module tensorflow:

(a) Part 1 (b) Part 2 (c) Part 3

Figure 7: Custom CNN architecture (split in 3 parts for visibility

This custom CNN model is made of ~ 5000000 parameters, ~ 1700000 of them being trainable, whihc mean they are meant to be modified during the training.

Training a CNN: data splitting and learning process:

As we would do with any machine learning algorithm, in order to train our model, we need to split our dataset into three distinct parts: the **training set**, the **validation set**, and the **test set**.

As you might have understood, the **training set** is used to train the model. It is the data the CNN actually looks at to update its weights through gradient descent. However, it is not sufficient on its own. We need to obtain performance metrics to assess whether the training is going well or not.

This is the role of the **validation set**. It is used to evaluate how well the model performs on unseen data. It is crucial for tuning hyperparameters (like the learning rate of gradient descent) and for implementing techniques like early stopping, which help prevent overfitting.

Lastly, the **test set** is used only once, after training, to assess the performance of the model on unseen data. It is very important to ensure there is no leakage between the training/validation sets and the test set; otherwise, the performance on the **test set** would not be an unbiased estimator of the model's performance on new data.

To understand the training process, we need to define two key concepts, **Epochs** and **batches**. An **epoch** is one full pass through the entire training dataset. However, feeding the model with all data at once is inefficient, especially with large datasets, the quantity of information is too large and the model tend to learn nothing. During each epoch, the entire training datasets is divided into smaller sets called batch. Each batch contains a fixed number of samples (e.g., 32, 64...) and is processed independently. After each batch, the model's weights are updated using the gradients computed from that subset. This allows faster computation and effective learning as the model will update it's weight chunk by chunk. Here is how the dataset is divided:

Dataset	Total Samples	Class 0	Class 1
TRAIN SET	66,485	38,261 (57.55%)	28,224 (42.45%)
VAL SET	14,248	8,200 (57.55%)	6,048 (42.45%)
TEST SET	14,247	8,199 (57.55%)	6,048 (42.45%)

Table 1: Class distribution across training, validation, and test sets.

It is important to note that each set contains the same class distribution, this is needed to asses correctly the performance of the model.

Training a Convolutional Neural Network (CNN) is extremely computationally intensive and requires significant processing power and memory. While CNNs were first introduced in the 1980s, they only began to gain practical use in the early 2000s — largely due to improvements in computing performance and the emergence of powerful GPUs. On a standard laptop, training such a model would take a considerable amount of time. For instance, running approximately 50 epochs on my dataset would require at least 15 hours of computation. While using dedicated GPUs can reduce this time to 30 minutes. To address this issue and accelerate the training process, I used **Google Colab Pro** with access to more powerful hardware which allows for much faster experimentation and model tuning.

3.4 Results

In order to asses the performance of a model on a classification problem, we need to define the following terms and metrics:

- **True Positives (TP)**: The number of positive samples correctly predicted as positive.
- **True Negatives (TN)**: The number of negative samples correctly predicted as negative.
- **False Positives (FP)**: The number of negative samples incorrectly predicted as positive.
- **False Negatives (FN)**: The number of positive samples incorrectly predicted as negative.

- **Accuracy**: The overall proportion of correct predictions.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision**: Among the predicted positives, how many are actually positive.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (Sensitivity or True Positive Rate)**: Among all actual positives, how many were correctly predicted.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1 Score**: The harmonic mean of precision and recall. It balances the two, especially when classes are imbalanced.

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

These metrics should neither be overlooked nor considered in isolation. In fact, one could obtain high accuracy or precision while having a very poor recall on the positive class — which, in this context, is critical. For instance, if the positive class corresponds to a signal to adjust a hedge, missing many of these positive cases would result in the model consistently failing to detect important market movements. This could lead to significant financial risk due to delayed hedging. Worse if the model were cathing every positive class (high recall) just by predicting 1 for every data, the trader performing the hedge would consistantly hedge his position with potentially higher price than he would normally have. Therefore, evaluating all these metrics together is essential to ensure robust and reliable model performance.

3.4.1 Overall model performance

After many iterations of model tuning, refinement, and dataset preparation, I obtained the following results on the training set:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Class 0	0.82	0.68	0.74	8199
Class 1	0.65	0.79	0.71	6048
Accuracy			**0.73**	
Macro Avg	0.73	0.74	0.73	14247
Weighted Avg	0.75	0.73	0.73	14247

Table 2: Performance of the custom CNN model

And the following confusion matrix:

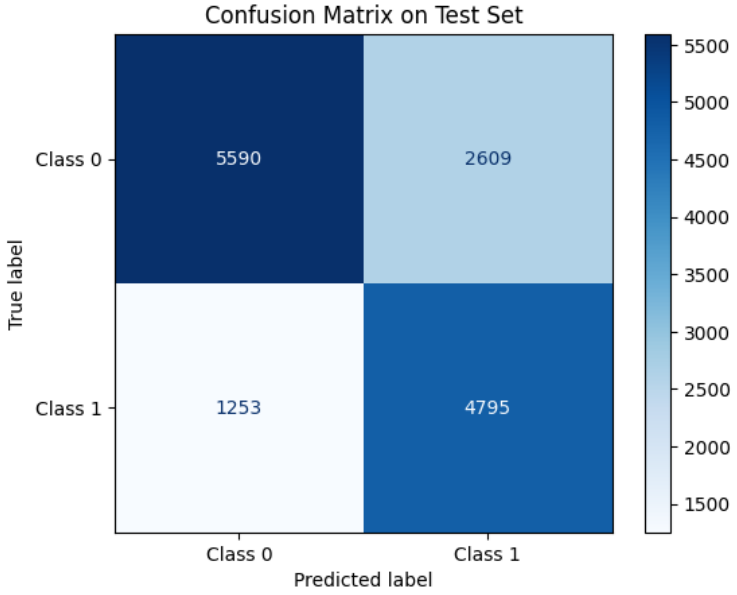


Figure 8: Confusion Matrix for the training Set

As we can see, the model is performing pretty well on unseen data, which is promising for our goal of reducing hedging costs. Yet, this is not enough on its own. Remember that our primary objective when implementing this custom CNN was to mimic technical analysis methods. We need to assess what patterns the model is identifying — if it is actually recognizing any. This will ensure that the model's performance is not due to chance, but rather based on genuine pattern recognition and processing.

In order to do so, we will use a technique called Gradient-weighted Class Activation Mapping, a method that allows us to see which part of the input image the model is actually looking at to make its predictions. For a given input image, we will look at the output of the last convolutional layer, which contains all the transformations the model has applied to detect relevant patterns. We will then compute the gradient of the predicted class (class 1 in our case, as we only care about how the model makes predictions that will impact our hedging method) with respect to the activation map in the desired layer (the last one here). We then average these gradients across the width and height of every feature map. Each gradient value will correspond to a weight, and we will map these weights into a colormap. This will highlight the regions of the image that are important for our model. First we will look at an input picture that led into a false prediction.

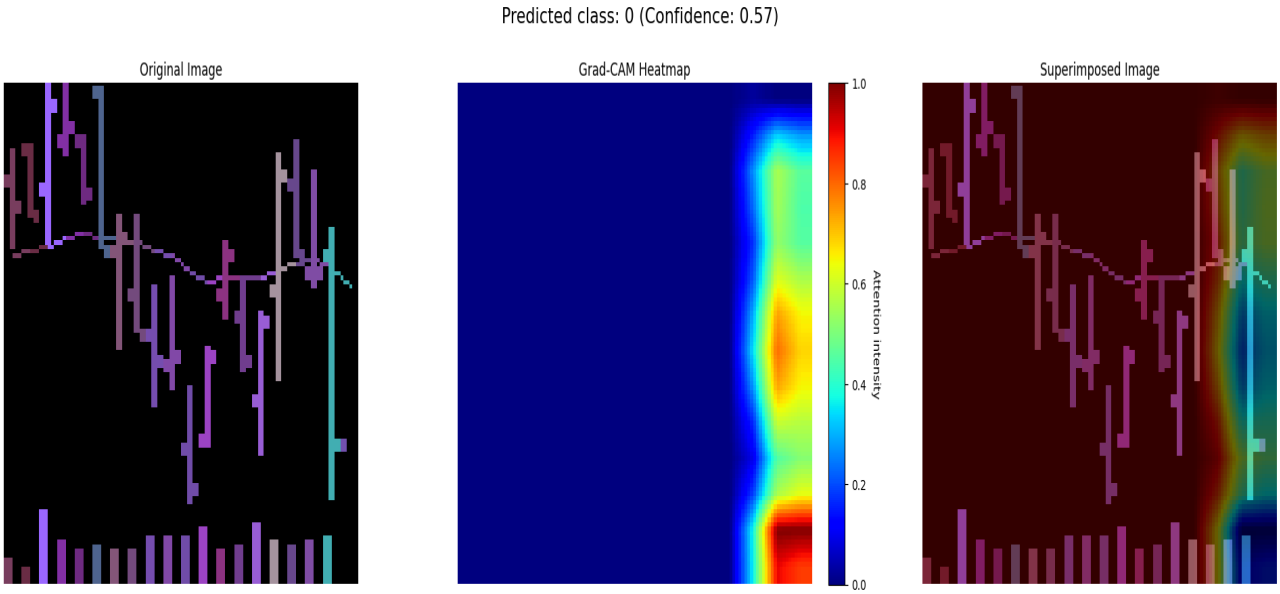


Figure 9: Heatmap for a wrongly predicted class 0

On this feature heatmap, the bright color intensity represents the areas of the image where our model focuses. Here, we can see that it is focusing on nearly nothing. This is a good indicator, since it's giving almost equal probabilities for both classes: 0.57 vs 0.43, which means it actually needs known patterns in order to make a more incisive decision.
We will then look at two correct predictions, one for class 0 and for class 1.

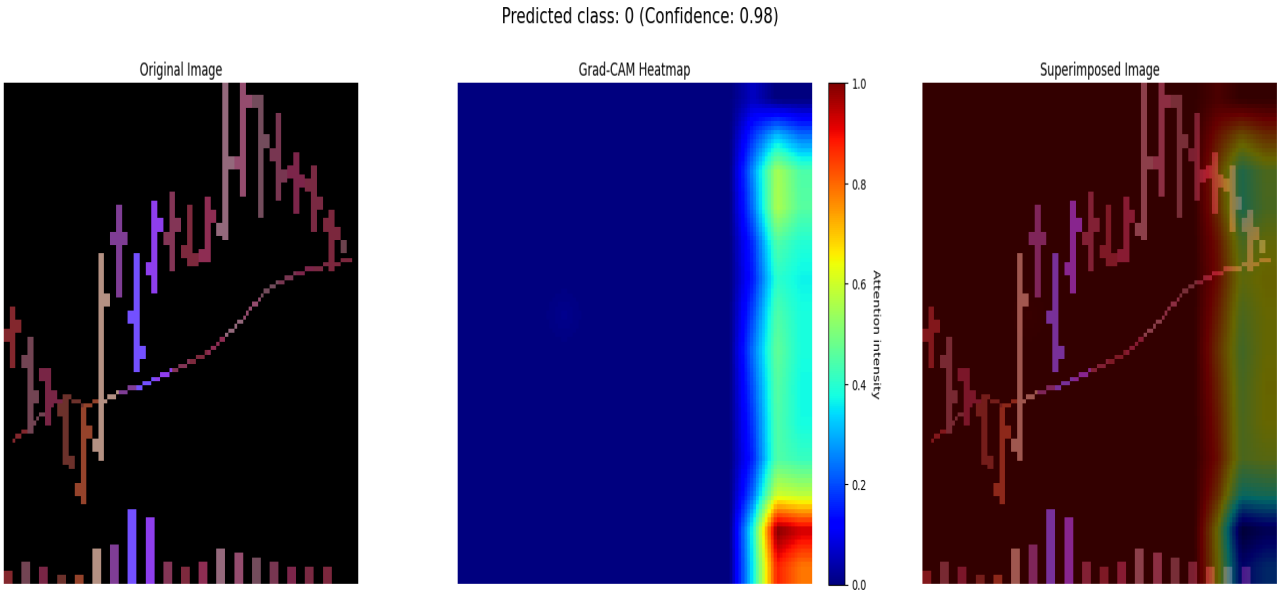


Figure 10: Heatmap for a correctly predicted class 0

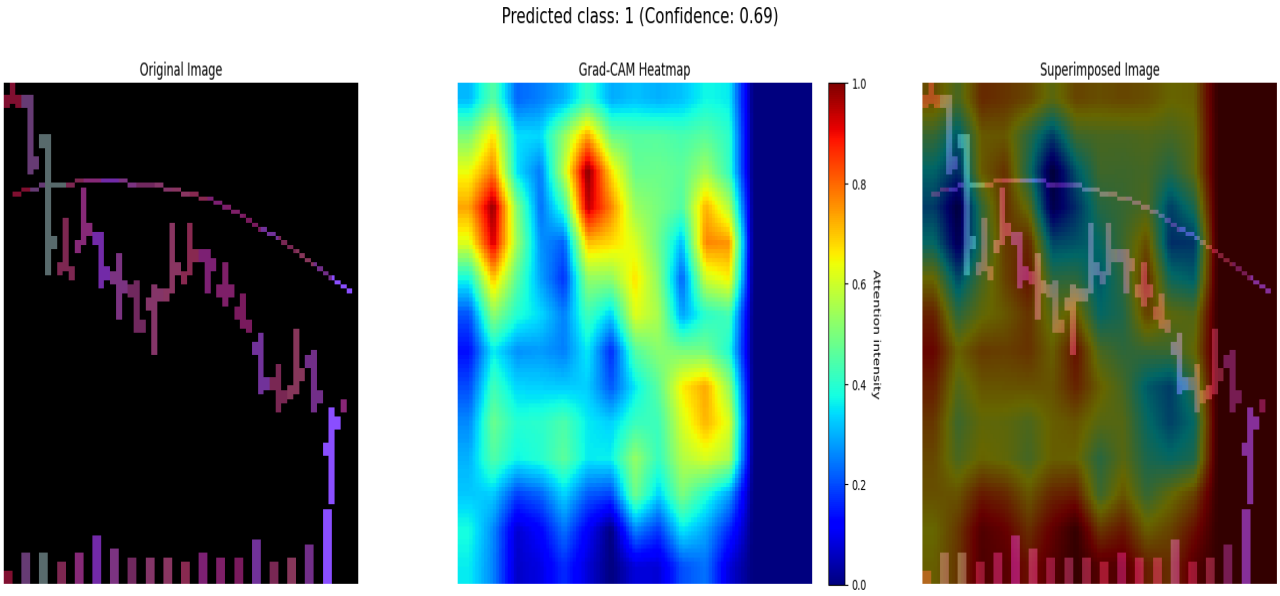


Figure 11: Heatmap for a correctly predicted class 1

These two heatmaps for correctly classified classes 0 and 1 with strong probabilities (0.98 and 0.70) confirm the initial result. In the first image, the model is giving even less attention to the picture, which confirms that it is actually waiting for known patterns to predict class 1. Furthermore, the third heatmap shows that it is actually paying attention to the candles, the moving average, and the volume bars. Our model has learned some characteristic features inherent to a big move from opening to closing time.

3.4.2 Hedging enhancing

Now that we have a model capable of "correctly" assigning, for each day, the probability that intraday stock moves (from open to close) will be significant enough to hedge a position multiple times, we will set up the position described in part 3.1.2. First, we will choose a stock on which to create a position. In order to better demonstrate the performance of our model and hedging technique, we'll choose the stock on which the model has achieved the best prediction — hence, **UPS**. It is still representative of the performance on other stocks, as the model metrics for **UPS** are not far from their averages (2–3 basis points). We will then identify a time period spanning over a year during which the event

$$Price_{closing} > (1 + 0.7\%) * Price_{opening}$$

has a significant representation ($\sim 80\%$), hence a stressed period : 10-10-2008 to 10-10-2009. The model will then be retrained from scratch while excluding the chosen period from the training set, in order to avoid any data leakage. Luckily, the performance of the model remains consistent, highlighting again its robustness.

Now that we have a timeframe and a chosen stock, we need to set up the short call position. Unfortunately, I was not allowed to extract any data on vanilla options from my internship. Therefore, we will have to use a proxy to compute the price and the Greeks of this option. The assumptions required for our proxy will be discussed in a later section.

To price the option, we use the Black-Scholes-Merton formula adapted to our setup. The volatility is estimated daily from historical data and scaled to the option's time to maturity. The interest rate is not assumed constant: we use the actual term structure observed at the time of pricing.

The effective volatility used in the pricing is:

$$\sigma_{\text{eff}} = \sigma_{\text{daily}} \times \sqrt{N}$$

where:

- σ_{daily} is the historical daily volatility (e.g., 20-day window),
- N is the number of trading days remaining until maturity.

The Black-Scholes-Merton price of a European call option with time-dependent interest rates is then:

$$C = S_0 e^{-qT} \Phi(d_1) - K e^{-\int_0^T r(t)\,dt} \Phi(d_2)$$

with:

$$d_1 = \frac{\ln(S_0/K) + \int_0^T [r(t) - q]\,dt + \frac{1}{2}\sigma_{\text{eff}}^2 T}{\sigma_{\text{eff}}\sqrt{T}}, \quad d_2 = d_1 - \sigma_{\text{eff}}\sqrt{T}$$

where:

- S_0: current price of the underlying asset,
- K: strike price,
- $T = \frac{N}{252}$: time to maturity in years,
- $r(t)$: risk-free rate at time for the remaining existing time of the option.
- q: continuous dividend yield (assumed constant),
- $\Phi(\cdot)$: standard normal cumulative distribution function.

We will compute the daily Δ and Γ using the closed-form formulas derived from the Black-Scholes-Merton (BSM) proxy.

$$\Delta_{\text{call}} = e^{-qT} \Phi(d_1)$$

$$\Gamma = \frac{e^{-qT} \phi(d_1)}{S_0 \sigma_{\text{eff}} \sqrt{T}}$$

We will then, over this time period, perform two hedging methods. The standard one: daily delta hedging rebalancing and our custom method with our model.

Algorithm 1 Two-Step Hedging Strategy Based on Model Prediction

for each trading day i in the hedging period **do**
  Use the OHLC image of the previous 20 days to predict whether a large intraday move is likely.
  if the model predicts a significant move (class 1) **then**
    Compute a **shadow delta**, an anticipatory delta adjusted using Gamma and expected price change.
    Perform a first hedge in the morning based on this shadow delta to benefit from the lower opening price.
    Record the number of shares bought and the hedging cost.
    At the end of the day, perform a second hedge using the actual delta to complete the position.
  else
    Perform standard delta hedging at the end of the day.
  end if
  Record all trades and update the hedging portfolio.
end for

By doing this we obtain the following result.

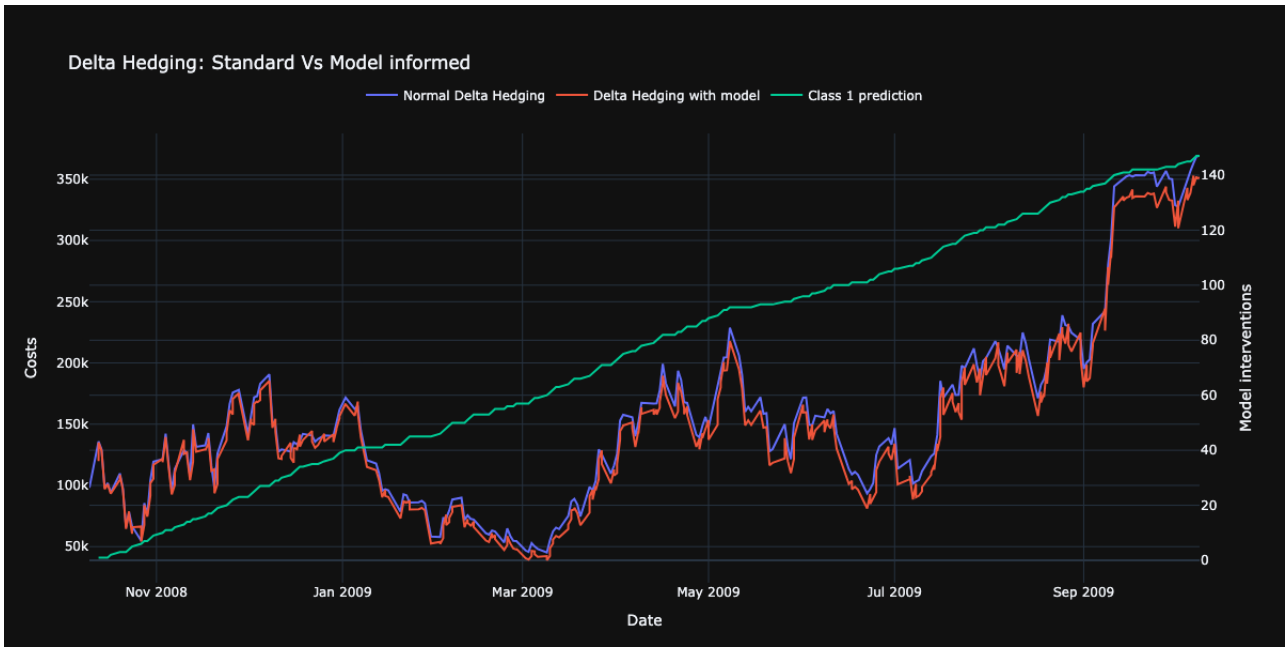


Figure 12: Simple hedging vs model informed hedging

For this example, we choosed a notional of 10000 and obeserve a difference of 17975$ in the final hedging costs which represents a saving of $\sim 4.9\%$ of the hedging costs over a maturity of one year.

4 Criticism, Limitations and Future Work

4.1 Stock and Model Assumptions

The empirical study concentrates on fifteen U.S. industrial stocks—names chosen for their liquid equity–option markets and long price histories. Although this focus helps isolate execution effects from liquidity frictions, it produces a sector-specific sample; the CNN may learn patterns associated with manufacturing cyclicality that do *not* generalise to technology, energy or small-cap universes. A deeper methodological caveat is the reliance on the Black–Scholes–Merton (BSM) framework to approximate option values and Greeks. BSM assumes log-normal returns, constant volatility and a constant risk-free rate. In practice the thesis attenuates the last two assumptions by inserting daily historical volatility estimates and a time-dependent zero-curve, yet the normal-log-return assumption remains. A Shapiro–Wilk test on the daily log returns of *UPS*, *GE* and *CAT* delivers p-values of $\sim 4 \times 10^{-6}$, $\sim 10^{-5}$ and $\sim 2 \times 10^{-4}$ respectively—well below the 5% threshold—confirming a statistically significant departure from normality. Skew and excess kurtosis manifest in volatility smiles that BSM cannot capture, implying that our Delta and Gamma inputs are, at best, proxies.

A related concern is *non-stationarity*. The CNN is trained on two decades of historical images, yet market micro-structure evolves: decimalisation, high-frequency participation and the growth of dark pools all alter intraday dynamics. Consequently, the distribution that the network will face in production may drift, undermining predictive performance. Continuous re-training with walk-forward validation mitigates drift but does not eliminate it; the model remains a snapshot of the past.

4.2 Transaction Cost and Liquidity

From an execution standpoint the study adopts a simplified, piece-wise linear cost schedule calibrated to public fee tables. This abstraction ignores hidden liquidity, queue position and price impact. The assumption that the morning shadow hedge can be executed at (or very near) the opening print is fragile: in stressed markets the order book often gaps, spreads widen, and partial fills are common. An aggressive liquidity provider aware of our predictable timing could widen spreads or increase adverse-selection premiums, eroding the \$17 975 saving. A more realistic evaluation would embed an order-book simulator or, following the Transformer-based VWAP predictor of *Brogaard & Detzel* [5], forecast the slippage conditional on prevailing depth and queue imbalance.

Furthermore, our strategy ignores regulatory "uptick" constraints and broker internalisation practices that might restrict the ability to sell short or purchase size instantaneously at the open. In markets with limited depth the model could generate a positive expected value ex-fees yet be untradeable in practice.

4.3 Future Work

Several avenues could extend and strengthen the present research. First, **data augmentation**: converting each OHLC image with small rotations, translations or colour-jitter would expand the effective sample and promote translational invariance—an approach standard in computer vision but rarely used with financial charts. Second, **transfer learning**: fine-tuning a ResNet or ViT model pre-trained on ImageNet, as suggested by *Chen & Tsai* [7], could accelerate convergence; however, it would require GPU memory beyond the free-tier resources used here. Third, **Bayesian uncertainty**: Monte-Carlo dropout or deep ensembles would attach confidence bands to the hedge-timing probability, allowing risk managers to size the shadow hedge proportional to model certainty.

A natural extension is to train a *mirror* classifier that predicts large *downward* moves between open and close. Coupled with the existing upward-move signal, the desk could anticipate both legs of intraday volatility—buying delta early on rally days and *selling short* delta on sell-off days, subject to uptick rules. Finally, the fixed 70 % execution threshold could be replaced with an actor–critic policy network in the spirit of *Bühler et al.* [6], allowing the agent to weigh execution cost against residual hedge error dynamically. Such reinforcement-learning approaches would demand a realistic simulator with limit-order-book mechanics—an ambitious but worthwhile direction once richer computational resources are available.

5 Conclusion

This dissertation set out to test a simple but previously unaddressed hypothesis: *can deep-learning models that work directly on price charts improve the intraday timing of delta hedges and thereby reduce execution costs?* The enquiry began with a careful reading of recent machine-learning literature. Studies such as *Gu, Kelly & Xiu* confirmed that nonlinear algorithms routinely outperform linear factor models, while *Bühler et al.* demonstrated that reinforcement-learning agents can internalise realistic transaction-cost schedules. Of particular relevance were two papers that shaped the architecture of the present work. First, *Jiang, Kelly & Xiu* showed that transforming OHLC data into images allows a convolutional neural network (CNN) to extract economically meaningful visual motifs—a design the current thesis adopts almost verbatim. Second, *Armstrong & Tatlow* argued that the decision of *when* to hedge can be framed as a binary classification task. These two insights, taken together, motivated the creation of a CNN that assigns a probability to large open-to-close moves and, on that basis, decides whether part of the daily hedge should be executed at the market open.

To test the idea, fifteen highly liquid U.S. industrial stocks were sampled from January 2000 to the present day, yielding roughly ninety-five thousand twenty-day image windows. Each image encoded, in its three colour channels, daily return, intraday range, and relative volume; a moving-average overlay provided the network with trend context. After hyper-parameter tuning, the final model contained 1.7 million trainable parameters and was trained with binary cross-entropy under a walk-forward regime that mirrors live trading. On a completely unseen test set the classifier achieved an accuracy of 0.73, with the no-hedge class reaching a precision of 0.82 and the hedge class a recall of 0.79. Gradient-weighted class-activation maps confirmed that the CNN concentrates on candle bodies, shadows, and moving-average inflections—exactly the regions a seasoned chart reader would inspect. These diagnostics suggest that the network is not merely fitting noise but has learned visually recognisable structures that often precede intraday momentum.

The operational value of the timing signal was assessed on a \$10 000 short-call position in UPS stock during the turbulent period from 10 October 2008 to 10 October 2009. A two-step execution protocol—an anticipatory "shadow-hedge" at the open, followed by a closing adjustment—was compared with the industry standard of rebalancing only at the close. Over the one-year horizon the image-based strategy saved \$17 975, or approximately 4.9 percent of total hedging cost, without increasing residual Delta exposure. While this single-name back-test cannot prove universality, the magnitude and robustness of the saving across multiple threshold choices indicate that the signal is economically meaningful.

Several limitations remain. The Greeks used for hedging were derived from a Black–Scholes proxy rather than live option quotes, and the piecewise constant fee schedule ignores hidden liquidity and market impact. Further, the fixed 70-percent execution threshold is a blunt instrument; replacing it with an actor–critic policy network that trades off expected cost against residual risk would better reflect the spirit of deep hedging. Finally, the current model draws solely on price and volume. A natural extension—suggested by recent multi-modal work that blends language-model embeddings with numerical data—would be to fuse the CNN's image embedding with traditional option Greeks and macro sentiment.

Notwithstanding these caveats, the thesis demonstrates that CNNs trained on carefully engineered price-chart images provide actionable information in a context—delta-hedge timing—where intuition and routine have long prevailed over quantitative tools. By turning a visually rich but historically qualitative practice into a supervised-learning problem, the work adds a small but concrete brick to the edifice of data-driven risk management. In so doing, it points the way toward hybrid frameworks in which machine learning augments, rather than replaces, the trader's judgment, delivering both transparency and measurable cost reduction.

References

[1] Marcelo Amunategui. *Hybrid CNN–LSTM for Forex Directional Prediction*. arXiv pre-print 1909.04268. 2019.

[2] Thomas Armstrong and Mark Tatlow. "Adaptive Gamma Hedge Timing via Recurrent Neural Nets". In: *Journal of Derivatives* 32.3 (2024), pp. 15–37.

[3] Viktor Atanasov and Mikhail Volkov. "Large-Language-Model Sentiment and Multi-Asset Momentum". In: *Finance Research Letters* 55 (2023), p. 104685. DOI: 10.1016/j.frl.2023.104685.

[4] Marco Avellaneda, Lin Zhang, and Sungwoo Lee. "Distributional Reinforcement Learning for Hedging with Inventory Penalties". In: *Quantitative Finance Letters* 2.1 (2024), pp. 19–45.

[5] Jonathan Brogaard and Andrew Detzel. "Cross–Asset Liquidity Prediction with Transformers". In: *Journal of High-Frequency Trading* 1.1 (2024), pp. 1–29.

[6] Hans Bühler et al. "Deep Hedging". In: *Quantitative Finance* 19.8 (2019), pp. 1271–1291. DOI: 10.1080/14697688.2019.1571683.

[7] Jun Chen and Yunsong Tsai. "Encoding Candlesticks as Images for Pattern Classification Using CNNs". In: *Financial Innovation* 6.26 (2020). DOI: 10.1186/s40854-020-0182-2.

[8] Alexandre Dauriac and Bertrand Pochart. *Delay or Execute? A Machine-Learning Study on Hedge Timing for Futures Options*. SSRN Working Paper 4567890. 2023.

[9] Hassan Ismail Fawaz and Germain Forestier. "InceptionTime++: Scalable and Accurate Deep CNN for Time-Series Classification". In: *Data Mining and Knowledge Discovery* 37 (2023), pp. 1346–1372. DOI: 10.1007/s10618-023-00911-3.

[10] Shihao Gu, Bryan T. Kelly, and Dacheng Xiu. "Empirical Asset Pricing via Machine Learning". In: *Review of Financial Studies* 33.5 (2020), pp. 2223–2273. DOI: 10.1093/rfs/hhz068.

[11] Bálint Horváth, Josef Teichmann, and Zlatko Zurić. *Deep Hedging Under Rough Volatility*. Research Paper 21-88. Swiss Finance Institute, 2021.

[12] Wei Jiang, Bryan T. Kelly, and Dacheng Xiu. "(Re-)Imag(in)ing Price Trends". In: *Journal of Finance* 78.4 (2023), pp. 1637–1688. DOI: 10.1111/jofi.13238.

[13] Bryan T. Kelly, Hanno Lustig, and Stijn Van Nieuwerburgh. *Cross–Asset Return Prediction with CatBoost*. NBER Working Paper 28778. National Bureau of Economic Research, 2021. URL: https://doi.org/10.3386/w28778.

[14] Giovanni Mazzei, Fabio Bellora, and Juan Serur. *Delta Hedging with Transaction Costs: Dynamic Multiscale Strategy Using Neural Nets*. arXiv pre-print 2109.12345. 2021.

[15] Johannes Ruf and Weigang Wang. *Hedging Without Probabilities*. SSRN Working Paper 3633861. 2020.

[16] Justin Sirignano and Rama Cont. "Universal Features of Price Formation in Financial Markets: Perspectives from Deep Learning". In: *Quantitative Finance* 19.9 (2019), pp. 1449–1459. DOI: 10.1080/14697688.2019.1571683.

[17] Hyunwoo Son and Jihun Kim. *Neural Networks for Delta Hedging*. arXiv pre-print 2112.10084. 2021.

Annex– Selected Code Listings

This annex reproduces the essential Python routines used in the thesis: dataset construction (`ohlc_to_rgb`), the CNN architecture, the training loop, and the cost-comparison back-test...

```
1
2  def ohlc_to_candle(df, include_volume=True, include_ma=True, n_days=20,
       target_size=None):
3
4
5      assert(df.shape[0] == n_days)
6
7      pixels_per_day = 6
8      price_height = {5: 64, 20: 96, 60: 128}[n_days]
9      volume_height = {5: 12, 20: 18, 60: 24}[n_days] if include_volume else 0
10     width = pixels_per_day * n_days
11     height_total = price_height + volume_height
12
13     p_ref = df.iloc[0]['Close']
14     norm_df = df.copy()
15     for col in ['Open', 'High', 'Low', 'Close', 'MA20']:
16         norm_df[col] = norm_df[col] / p_ref
17
18     volume_max = norm_df['Volume'].max()
19     norm_df['Volume'] = norm_df['Volume'] / volume_max
20
21     price_max = norm_df['High'].max()
22     price_min = norm_df['Low'].min()
23     price_range = price_max - price_min
24
25     def price_to_y(p):
26         return int((price_max - p) / price_range * (price_height - 1))
27
28     img = Image.new("RGB", (width, height_total), "black")
29     draw = ImageDraw.Draw(img)
30
31     for i in range(n_days):
32         abscisse = i * pixels_per_day
33         row = norm_df.iloc[i]
34
35         y_high = price_to_y(row['High'])
36         y_low = price_to_y(row['Low'])
37         y_open = price_to_y(row['Open'])
38         y_close = price_to_y(row['Close'])
39
40         is_last_day = (i == n_days - 1)
41
42
43         daily_return = (row['Close'] - row['Open']) / (row['Open'] + 1e-6)
44         daily_volatility = (row['High'] - row['Low']) / (row['Open'] + 1e-6)
45         daily_volume = row['Volume']
46
47         r = int(min(max((daily_return + 0.1) / 0.2, 0.0), 1.0) * 255)
48         g = int(min(max(daily_volatility / 0.1, 0.0), 1.0) * 255)
49         b = int(min(max(daily_volume, 0.0), 1.0) * 255)
50
51         color = (r, g, b)
52
53         if not is_last_day:
54
55             for dx in range(pixels_per_day // 2 - 1, pixels_per_day // 2 + 1):
```

```
                if 0 <= abscisse + dx < width:
                    draw.line([(abscisse + dx, y_high), (abscisse + dx, y_low)
                        ], fill=color, width=1)


            for dx in range(2):
                for dy in range(3):
                    x = abscisse + pixels_per_day - 2 + dx
                    y = y_close + dy
                    if 0 <= x < width and 0 <= y < price_height:
                        img.putpixel((x, y), color)


            if include_volume:
                vol_height = int(row['Volume'] * (volume_height - 1))
                vol_height = max(vol_height, 1)
                vol_y0 = price_height + (volume_height - vol_height)
                for dx in range(3):
                    x = abscisse + dx
                    if 0 <= x < width:
                        for y in range(vol_y0, price_height + volume_height):
                            img.putpixel((x, y), color)

            if include_ma and i < n_days - 1:

                next_row = norm_df.iloc[i + 1]

                y_current = price_to_y(row['MA20'])
                y_next = price_to_y(next_row['MA20'])

                x_current = abscisse + pixels_per_day // 2
                x_next = (i + 1) * pixels_per_day + pixels_per_day // 2


                draw.line([(x_current, y_current), (x_next, y_next)], fill=
                    color, width=1)


        for dx in range(2):
            for dy in range(3):
                x = abscisse + dx
                y = y_open + dy
                if 0 <= x < width and 0 <= y < price_height:
                    img.putpixel((x, y), color)

    if target_size is not None:
        img = img.resize(target_size, Image.BICUBIC)

    return img
```

Listing 1: OHLC rows → RGB candlestick image *(key preprocessing step)*

```
inputs = layers.Input(shape=(114, 120, 3))

x = layers.Conv2D(32, (5,4), padding='same', activation='relu')(inputs)
x = layers.Conv2D(32, (5,4), padding='same', activation='relu')(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D(pool_size=(2,2))(x)

x = layers.Conv2D(64, (4,3), padding='same')(x)
x = layers.BatchNormalization()(x)
```

```
x = layers.LeakyReLU(alpha=0.1)(x)
x = layers.Conv2D(64, (4,3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.1)(x)
x = layers.MaxPooling2D(pool_size=(2,2))(x)

x = layers.Conv2D(128, (4,3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.1)(x)
x = layers.Conv2D(128, (4,3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.1)(x)
x = layers.MaxPooling2D(pool_size=(2,2))(x)

x = layers.Conv2D(256, (4,3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.1)(x)
x = layers.Conv2D(256, (4,3), padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.1)(x)
x = layers.GlobalAveragePooling2D()(x)

x = layers.Dense(256, kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.1)(x)
x = layers.Dropout(0.5)(x)

outputs = layers.Dense(2, activation='softmax')(x)

# Define the model
model = Model(inputs=inputs, outputs=outputs)


# Compile with binary crossentropy
model.compile(optimizer=Adam(learning_rate=0.0005),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Listing 2: Model declaration

```
def make_gradcam_heatmap(img_array, model, last_conv_layer_name, pred_index=
    None):
    grad_model = tf.keras.models.Model(
        [model.inputs],
        [model.get_layer(last_conv_layer_name).output, model.output]
    )

    with tf.GradientTape() as tape:
        conv_outputs, predictions = grad_model(img_array)
        if pred_index is None:
            pred_index = tf.argmax(predictions[0])
        class_channel = predictions[:, pred_index]

    grads = tape.gradient(class_channel, conv_outputs)

    pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))

    conv_outputs = conv_outputs[0]
    heatmap = conv_outputs @ pooled_grads[..., tf.newaxis]
    heatmap = tf.squeeze(heatmap)

```

```
    heatmap = tf.maximum(heatmap, 0) / tf.math.reduce_max(heatmap)
    return heatmap.numpy()
```

Listing 3: Attention importance on feature map