# Comparison of deep learning algorithms for forecasting stock returns and portfolio optimization

**Sarah Witzman**
switzman@princeton.edu

**Mathieu-Savvas Dimitriades**
md0101@princeton.edu

**Keywords:** Deep Learning, Neural Networks, Stock Returns Predictability, Time Series Forecasting, Performance Evaluation Criteria

**Abstract**

In this paper, we explore several deep learning algorithms applied to the prediction of stock returns of stocks in the CAC40, the French stock index. The algorithms we consider are: **Multi-layer perceptron, Convolutional, Recurrent (LSTM, GRU) and Transformer neural network architectures**. Since the main objective of market practitioners is as much to predict the effective return of an asset than to predict its sign [2], we equally focus on two tasks: prediction of stock returns and binary classification of the sign of those returns (+ / -) and compare the performance of the two approaches. For comparison, we will also consider predictions obtained using linear regression and logistic regression. An important aspect we analyze is the performance criteria used to assess these algorithms [2]. Once the forecasts of our models obtained, we propose the use of Enhanced Portfolio Optimization [16] to combine these forecasts into a trade strategy. This enables us to evaluate the **performances of the investment strategies considered against those of the CAC40 index**.

# 1 Introduction

Classical econometrics methods are commonly used in finance to analyze financial data and model relationships between variables. However, most of these methods are linear, meaning they assume that the relationship between variables is linear in its parameters. Although most of the research dealing with the predictability of stock returns favors the linear framework, there is a growing strand of empirical research that highlights the relevance of nonlinear methods[1]. The main reason for this is that linear methods have limitations when it comes to predicting asset returns which have been proven to be highly nonlinear. This is because returns are influenced by various factors such as economic conditions, geopolitical events, market sentiment, and investor behavior, which may not be accurately captured by linear relationships. Moreover, financial markets are subject to sudden shocks and extreme events, such as market crashes or black swan events, which can disrupt the linear relationships and render linear models ineffective in predicting returns. This inherent randomness and uncertainty of financial markets make it difficult to accurately predict returns using linear models, as even small changes in input variables or assumptions can result in significant differences in predicted outcomes. This is why many financial researchers are turning to deep learning and non-linear models to develop more accurate and effective predictive models which can capture non-linear relationships between variables.

This paper explores the application of deep-learning techniques to predicting stock market returns. More specifically, we will try to forecast 1-day ahead returns of stocks in the CAC40 index using several deep-learning algorithms trained on three types of features: fundamental, macroeconomic and technical [3]. **Can deep learning techniques effectively forecast future returns from past information? How can we build, based on those forecast, a tradable portfolio? Can this portfolio outperform the market? How can we evaluate the quality of a given model?** Those are the questions we want to tackle in this study.

The paper is organized as follows: Part II discusses related work. Part III discusses the data used in the paper and the data preparation process. Part IV highlights the learning algorithms used. Part V focuses on combining model forecasts into a tradable portfolio. Part VI discusses the performance of the various algorithms under various metrics. Part VII analyzes further directions to the paper. Part VIII concludes.

# 2 Related work

To the best of our knowledge, this is the first paper to specifically benchmark various deep learning algorithms and compare their performance based on the profit and loss of a strategy which combines the forecasts given by those models using portfolio optimization [16].

[3] propose a deep learning methodology based on long-short term memory (LSTM) to forecast next day returns of the S&P500 index. [21] propose a deep learning framework that combines stacked auto-encoders and LSTM models to predict stock returns. [22] used a deep learning model called WaveNet to predict stock price trends based on daily stock price data. [23] used a deep learning model with both LSTM and GRU layers to predict stock prices based on both numerical and textual information from financial news articles. [26] use return data and news sentiment analysis with various combinations of network architectures including MLP and LSTM. [27] propose an alternate combined LSTM/GRU architecture with both news sentiment analysis and a deep auto-encoder. [24] evaluate the performance of MLP, simple RNN, and LSTM models in predicting the returns of 4 stock groups within the Tehran stock exchange, finding that LSTM performed the best but also had the longest runtime. [25] compared MLP and CNN frameworks in forecasting the overall performance of the Indian National Stock Exchange.

However, the combination of the forecasts into a trade strategy was either non-existent ([3], [22], [24], [25], [26], [27]) or based on

the simplistic criteria of going long the stock if the predicted returns are positive and going short the stock if the predicted returns are negative ([21], [23]). We want to further study the relevance of combining the forecasts using portfolio optimization in order to be able to compare the models considered in terms of both profitability and predictive capabilities. In addition to analyzing porfolio construction, we also aim to more thoroughly explore the performance of different models by directly comparing MLP, CNN, LSTM, GRU, and TFT architectures for the same dataset and optimization task.

# 3 Data

Our signal construction methodology is based on the work of [3] and we adapt their signals to the French stock market. We consider **fundamental, macroeconomic and technical data**. For each stock, the target variable is the one-day ahead forecast of the return of the stock in the case of prediction and the one-day ahead forecast of the sign of the return in the case of classification. The fundamental data we use is the lagged returns series. The macroeconomic data consists of the CBOE VIX index, the EUR/USD exchange rate, French unemployment rate and consumer sentiment index. The technical indicators consist of MACD, ATR and RSI.

## 3.1 Signals used

The data used spans from the $1^{st}$ on June, 2009 to the $29^{th}$ of March, 2023. For each stock $i$ in the CAC40 index (we drop the subscript $i$ here for conciseness), we use the following:

**Fundamental data:**
**Log-returns**: $r_t = ln(\frac{P_t}{P_{t-1}})$ (source: Yahoo Finance)

**Macroeconomic data:**
**Returns of the CAC40 index** at time t ($FCHI_t$) (source: Yahoo Finance)
**CBOE VIX index** ($VIX_t$) (source: Yahoo Finance)
**Euro-USD exchange rate** ($EURUSD_t$) (source: Yahoo Finance)
**French Unemployment Rate** ($UR_t$) (source: FED St. Louis)
**French Consumer Sentiment Index** ($CSI_t$) (source: FED St. Louis)

**Technical signals:**
**Standardized k-days returns**: $\forall k \in \{5, 10, 15, 20, 25, 30\}, r_t^k = \frac{1}{k} ln(\frac{P_t}{P_{t-k}})$
**Average True Range** at time t: $ATR(n)_t = \frac{\sum_{k=t-n}^{t} max(|H_k - L_k|, |H_k - P_{k-1}|, |L_k - P_{k-1}|}{t}$ for $n = 14$
**Relative Strength Index**: $RSI(n)_t = 100.(1 - (1 + \frac{\sum_{i=t-n}^{t} r_i I(r_i > 0)}{\sum_{i=t-n}^{t} r_i I(r_i < 0)})^{-1})$ for $n = 14$
**Moving Average Convergence Divergence**: $MACD_t = EMA(r_t, 12) - EMA(r_t, 26)$

The rationale for ATR is that a stock experiencing a high level of volatility has a higher ATR, and a lower ATR indicates lower volatility for the period evaluated. RSI is a technical indicator used in trading to measure the strength of price action in a particular financial asset. MACD is a trend signal that is calculated by subtracting the 26-day exponential moving average (EMA) from the 12-day EMA. We include k-days returns momentum signals to capture the trend of the returns of the stock.

## 3.2 Data preparation

The first step in our data preparation process involves making sure that our data does not contain too many missing values. To avoid the necessity to use imputation techniques, we drop the stocks with more than 80% missing values. These stocks are Worldline (WLN.PA) and Stellantis (STLAP.PA). Beyond that, we align the signals for the different stocks, by increasing the frequency of the monthly signals to monthly-constant daily signals.

To feed our data through the various neural networks, we then proceed to normalize the data using standard scaling. Our statistics are computed on the training set, which expands from 2009 to 2014. The whole data set is then normalized by standard scaling i.e. the function $f(x) = \frac{x - E_{train}(x)}{SD_{train}(x)}$ is applied on the data set, column-wise. Note that all the statistics are computed on the train set to avoid forward-looking bias.

At the end of this step, our data set consists of 3344 observations and 608 columns (16 columns for each stock). In practice, we are going to use, for each stock, its 14 predictors to build our daily forecasts of the stock return and the sign of the stock return.

# 4 Deep-learning algorithms considered

## 4.1 Multi-layer perceptron

The simplest deep networks are called multilayer perceptrons (MLP) [4], and they consist of multiple layers of neurons each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence).

We denote by the matrix $X \in \mathrm{R}^{n \times d}$ a minibatch of $n$ examples where each example has d inputs (features). For a one-hidden-layer MLP whose hidden layer has $h$ hidden units, we denote by $H \in \mathrm{R}^{n \times h}$ on the outputs of the hidden layer, which are hidden representations. Since the hidden and output layers are both fully connected, we have hidden-layer weights $W^{(1)} \in \mathrm{R}^{d \times h}$ and biases $b^{(1)} \in \mathrm{R}^{1 \times h}$ and output-layer weights $W^{(2)} \in \mathrm{R}^{h \times q}$ and biases $b^{(2)} \in \mathrm{R}^{1 \times q}$. This allows us to calculate the outputs of the one-hidden-layer MLP as follows:

$$\mathbf{H} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$$

$$\mathbf{O} = \sigma_2(\mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$$

$(\sigma_i)_{i \in (1,2)}$ are activation functions. In practice, the most commonly used activation functions are ReLU (Rectified Linear Unit) defined as $ReLU(x) = max(x, 0)$, $sigmoid(x) = \frac{1}{1+e^{-x}}$, hyperbolic tangent defined as $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. It is important to note that, depending on the target variable we wish to predict, $\sigma_2$ is going to have a determinant role in making sure we are mapping to the correct output interval.

The strength of perceptrons lies in the Universal Approximation theorem [5]: even with a single-hidden-layer network, given enough nodes and the right set of weights, we can model any function.

## 4.2 Convolutional neural network

Convolutional neural networks are a class of neural networks that use a mathematical operation called convolution in place of general matrix multiplication in at least one of their layers [6]. CNNs are

regularized versions of MLP. The full connectivity of the latter networks make them prone to over-fitting data. Typical ways of regularization, or preventing over-fitting include: penalizing parameters during training (weight decay) or trimming connectivity (skipped connections, dropout...). CNNs take advantage of the hierarchical pattern in data and assemble patterns of increasing complexity using simpler patterns implied by their filters.

**Convolutions** The convolution of two functions $f, g \in \mathrm{R}^d \to \mathrm{R}$ is defined as: $(f*g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x}-\mathbf{z})d\mathbf{z}$. That is, we measure the overlap between $f$ and $g$ when one function is flipped and shifted by $\mathbf{x}$. When we have discrete objects, the integral turn into a sum. In practice, convolutional neural-networks use cross-correlation instead of convolutions, i.e. we flip and shift one function by $-\mathbf{x}$ instead.

We will be applying CNNs in the context of 1D-vectors. We denote by the matrix $X \in \mathrm{R}^{n\times d}$ a mini-batch of $n$ examples where each example has d inputs (features). Let $k \in \mathbf{N}$ be the kernel size and $N \in \mathbf{N}$ be the number of channels considered. The convolutional layer operates as follows: $\forall i \in [1,n], \forall j \in [1,N], y_i^j = b^j + w^j * X^i$, where $(w^j * X^i)_m = \sum_{p=-\Delta}^{\Delta} w_p^j X_{m+p}^i$ and $\Delta$ is the half-kernel size. The output of the single-layer convolutional neural network is then obtained by applying an MLP to the output of the activated convolutional layer.

## 4.3 Recurrent neural network

Up until now, we have been making the underlying assumption that the data points $(\mathbf{X_i}, y_i)_{i\in[1,n]}$ are independent and identically distributed. However, in our case, a possible way to improve our models is to account for the fact that, at time $t$, our prediction may depend on what happened at time $t-k, t-k+1, ..., t-1$ where $k \in \mathbf{N}$. Therefore, we introduce a new class of neural networks, called Recurrent Neural Networks, which take into account this time-series dependence. The aforementioned k, in the context of recurrent neural networks, is called the *look-back parameter*.

RNNs are latent variables models, where the latent variables store information up to the previous time-step. They build on the Back-Propogation Through Time algorithm [9], which requires us to expand (or unroll) the computational graph of an RNN. The unrolled RNN is essentially a FFN with the property that the same parameters are repeated throughout the unrolled network, appearing at each time step. Then, just as in any feedforward neural network, we can apply the chain rule, backpropagating gradients through the unrolled net. The gradient with respect to each parameter must be summed across all places that the parameter occurs in the unrolled net. This leads to complications because sequences can be rather long, leading to vanishing and exploding gradient issues [10]. Two fixes to the vanishing / exploding gradient where proposed with the Long-Short Term Memory (LSTM) and Gated Reccurrent Unit (GRU) architectures on which we will focus in our study.

### 4.3.1 Long-Short Term Memory (LSTM)

LSTMs resemble standard recurrent neural networks but here each ordinary recurrent node is replaced by a memory cell. Each memory cell contains an internal state, i.e., a node with a self-connected recurrent edge of fixed weight 1, ensuring that the gradient can pass across many time steps without vanishing or exploding.

Mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d. Thus, the input is $X_t \in \mathrm{R}^{n\times d}$ and the hidden state of the previous time step is $H_{t-1} \in \mathrm{R}^{n\times h}$. Correspondingly, the gates at time step $t$ are defined as follows: the input gate is $I_t \in \mathrm{R}^{n\times h}$, the forget gate is $F_t \in \mathrm{R}^{n\times h}$, and the output gate is $O_t \in \mathrm{R}^{n\times h}$. They are calculated as follows:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i)\\ \mathbf{F}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f)\\ \mathbf{O}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o)\end{aligned}$$

The input node is calculated as follows: $\tilde{\mathbf{C}}_t = \mathbf{tanh}(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$. $\mathbf{W}_{xi}, \mathbf{W}_{hi}, \mathbf{W}_{xf}, \mathbf{W}_{hf}, \mathbf{W}_{xo}, \mathbf{W}_{ho}$ are weight parameters. $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o, \mathbf{b}_c$ are biais parameters.

The input gate governs how much we take new data into account via the input node, and the forget gate addresses how much of the old cell internal state we retain. Using the Hadamard (elementwise) product operator we arrive at the following update equation: $\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$. The hidden state is then updated as: $\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$.

Whenever the output gate is close to 1, we allow the memory cell internal state to impact the subsequent layers uninhibited, whereas for output gate values close to 0, we prevent the current memory from impacting other layers of the network at the current time step.

### 4.3.2 Gated Recurrent Unit (GRU)

The gated recurrent unit (GRU) [12] offers a simpler version of the LSTM memory cell that often achieves comparable performance but with the advantage of being faster to compute. The LSTM's three gates are replaced by two: the reset gate and the update gate.

Mathematically, for a given time step $t$, we denote by $\mathbf{X}_t \in \mathrm{R}^{n\times d}$ and the hidden state of the previous time step is $H_{t-1} \in \mathrm{R}^{n\times h}$. Then the reset gate $R_t \in \mathrm{R}^{n\times h}$ and update gate $Z_t \in \mathrm{R}^{n\times h}$ are computed as:

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r)\\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z)\end{aligned}$$

The candidate hidden state is then given by: $\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\,\mathbf{W}_{hh} + \mathbf{b}_h)$.

Compared with a vanilla RNN model where $\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$ for a given activation function $\phi$, weights $\mathbf{W}_{xh}, \mathbf{W}_{hh}$ and bias $\mathbf{b}_h$, now the influence of the previous states can be reduced with the element-wise multiplication of $\mathbf{R}_t$ and $\mathbf{H}_{t-1}$. When the reset gate is close to 1, we get a vanilla RNN. When it is close to 0, we get an MLP with $\mathbf{X}_t$ as input.

The hidden state is updated as: $\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1-\mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$. Whenever the update gate is close to 1, we simply retain the old state. In this case, we are effectively skipping time step $t$ as information from $\mathbf{X}_t$ is ignored. Whenever the update gate is close to 0, the new latent state approaches the candidate latent state.

## 4.4 Transformer

The last architecture we consider is the transformer architecture [8]. Transformers are able to capture long-range dependencies and interactions, a feature very attractive for time series modeling. The core idea behind the transformer model is the attention mechanism.

**Attention mechanism** The attention over a set $\mathbf{D}$ is defined as: $\mathbf{D} = \{(\mathbf{k}_1, \mathbf{v}_1), ..., (\mathbf{k}_m, \mathbf{v}_m)\}$ of (key, value) pairs is defined as follows: $\mathbf{f}(\mathbf{q}, \mathbf{D}) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ where $\forall i \in (1, ..., m), \alpha(\mathbf{q}, \mathbf{k}_i) \in \mathrm{R}$ [8]. $\alpha$ is a kernel function. For the Gaussian kernel, we have: $\alpha(\mathbf{q}, \mathbf{k}) = \exp\left(-\frac{1}{2}\|\mathbf{q}-\mathbf{k}\|^2\right)$. When queries $\mathbf{q}$ and keys $\mathbf{k}$ are of

different dimensionalities, we use an additive attention function defined as follows: $\alpha(\mathbf{q},\mathbf{k}) = \mathbf{w}_v^T tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathrm{R}$, where $\mathbf{q} \in \mathrm{R}^q, \mathbf{k} \in \mathrm{R}^k$, are the vector of queries, keys and $\mathbf{W}_q \in \mathrm{R}^{h\times q}$, $\mathbf{W}_k \in \mathrm{R}^{h\times k}$ and $\mathbf{w}_v \in \mathrm{R}^h$ are trainable parameters. When queries and keys are of the same dimension d, we can use scaled-dot product attention function: $\alpha(\mathbf{q},\mathbf{k}_i) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i/\sqrt{d})}{\sum_{j=1}\exp(\mathbf{q}^\top \mathbf{k}_j/\sqrt{d})}$.

**Multi-head attention** In practice, given the same set of queries, keys, and values we may want our model to combine knowledge from different behaviors of the same attention mechanism. Instead of performing a single attention pooling, queries, keys, and values can be transformed with independently learned linear projections. Then these projected queries, keys, and values are fed into attention pooling in parallel. In the end, attention pooling outputs are concatenated and transformed with another learned linear projection to produce the final output. This design is called *multi-head attention* [8], where each of the attention pooling outputs is a head. For a query $\mathbf{q} \in \mathrm{R}^{d_q}$, a key $\mathbf{k} \in \mathrm{R}^{d_k}$, and a value $\mathbf{v} \in \mathrm{R}^{d_v}$, we compute each attention head $h_i$ as:

$$\forall i \in (1,...,h), \mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in R^{p_v} \tag{1}$$

$\mathbf{W}_i^{(q)} \in \mathrm{R}^{p_q\times d_q}$, $\mathbf{W}_i^{(k)} \in \mathrm{R}^{p_k\times d_k}$ and $\mathbf{W}_i^{(v)} \in \mathrm{R}^{p_v\times d_v}$ are learnable parameters, $f$ is an attention pooling function. The multi-head attention output is another linear transformation via learnable parameters of the concatenation of $\mathbf{h}$ heads: $\mathbf{W}_o \in \mathrm{R}^{p_o\times hp_v}$: $\mathbf{W}_o \begin{bmatrix}\mathbf{h}_1\\ \vdots \\ \mathbf{h}_h\end{bmatrix} \in \mathrm{R}^{p_o}$.

**Self-attention** Given a sequence $\mathbf{x}_1,\ldots,\mathbf{x}_n$, ($x_i \in \mathrm{R}^d$), and an attention pooling function $f$, its self-attention outputs are defined as:

$$\forall i \in (1,...,n),\ \mathbf{y}_i = f(\mathbf{x}_i,(\mathbf{x}_1,\mathbf{x}_1),...,(\mathbf{x}_n,\mathbf{x}_n)) \in \mathrm{R}^d$$

**Positional encoding** Unlike RNNs, which recurrently process tokens of a sequence one by one, self-attention does not rely on sequential operations to enable parallel computation. Therefore, self-attention does not preserve the order of the sequence. To preserve information about the order of tokens, we can include a *positional encoding feature* as an additional input associated with each token.

**Transformer architecture** The transformer is composed of an encoder and a decoder. The transformer encoder is a stack of multiple identical layers, where each layer has two sub-layers. The first is a multi-head self-attention pooling and the second is a position-wise feed-forward network. Specifically, in the encoder self-attention, queries, keys, and values are all from the outputs of the previous encoder layer. The transformer decoder is also a stack of multiple identical layers with residual connections and layer normalizations [13]. Besides the two sublayers described in the encoder, the decoder inserts a third sublayer, known as the encoder-decoder attention, between these two. In the encoder-decoder attention, queries are from the outputs of the previous decoder layer, and the keys and values are from the transformer encoder outputs. In the decoder self-attention, queries, keys, and values are all from the outputs of the previous decoder layer. However, each position in the decoder is allowed to only attend to all positions in the decoder up to that position. This masked attention preserves the auto-regressive property, ensuring that the prediction only depends on those output tokens that have been generated.

**Time-Fusion Transformer** In this paper, we consider a specific transformer called Time-Fusion Transformer [14] which main features are the following:

1. **Gating mechanisms:** These mecanisms discard unused components based on: (a) Gated Linear Units (GLU) defined as: $GLU(\gamma) = \sigma(\mathbf{W}_1\gamma+\mathbf{b}_1)\odot\mathbf{W}_2\gamma+\mathbf{b}_2$, the weights being shared across all gates (b) Gated Residual Networks (GRN) defined as: $GRN(\mathbf{a},\mathbf{c}) = LayerNorm(\mathbf{a} + GLU(\mathbf{W}_3 ELU(\mathbf{W}_4\mathbf{a} + \mathbf{W}_5\mathbf{c} + \mathbf{b}_3) + \mathbf{b}_4)$, the weights being shared across all time steps (LayerNorm was introduced in [28] and is defined as: $LayerNorm(x) = \frac{x-E(x)}{\sqrt{Var(x)+\varepsilon}}\gamma + \beta$ where $\gamma,\beta$ are learnable parameters).
2. **Variable selection networks:** These networks select relevant input variables at each time step by using a GRN for each variable, whose weights are shared across all time steps.
3. **Static covariate encoders:** These encoders integrate specific features to the network using four types of context vectors (encoded using GRN) for temporal variable selection, local processing of temporal features and enriching of temporal features with static information.
4. **Interpretable multi-head attention:** Instead of using the multi-head attention as defined in (1) in which each has different value weights, TFT uses the same weights for all the values as follows:

$$\forall i \in (1,...,h), \mathbf{h}_i = \frac{1}{h}\sum_{i=1}^{h} f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}^{(v)}\mathbf{v}) \in R^{p_v} \tag{2}$$

5. **Temporal processing:** This feature enables to learn both short and long-term temporal relationships for both observed and known time-varying inputs (local processing is handled by a sequence-to-sequence layer which replaces positional encoding described above and global processing is handled by interpretable multi-head attention described above).
6. **Prediction intervals:** Prediction intervals are constructed via quantile forecasts to determine the range of the target values at each forecast horizon.

# 5 Portfolio construction

As described in the introduction, each of our prediction algorithms will try to forecast the returns or the sign of those returns. Although we could use the same algorithm for both tasks, simply taking the sign of the forecasted returns as a forecast for the sign for the returns, our approach is to use regression models to forecast returns and classification models to forecast the sign of these returns. This is because we believe that the increased simplicity of the second task may lead to better prediction performance overall. For each of these tasks, we propose a portfolio construction methodology.

## 5.1 Return forecast

In a regression setting, all of the algorithms described above are designed to compute, for each of the $n$ assets in our portfolio, the vector of conditional excess returns of these assets: $\forall i \in (1,...,n), \alpha_t^i =$

$E(\mathbf{r}_t^i - \mathbf{r_{f_t}} | (\mathbf{S}_{t-1}^i, ..., \mathbf{S}_{t-k}^i))$ based on a multidimensional signal time-series $\mathbf{S} \in \mathrm{R}^{T \times n \times d}$, $r_f$ being the risk-free rate time-series and $k \in \mathrm{N}$ the *look-back window*. Similarly, we would like to compute the conditional covariance matrix $\mathbf{\Sigma}_{t-1} = \mathbf{V}(\mathbf{r}_t | \mathcal{F}_{t-1})$ where $\mathcal{F}_{t-1}$ is the information set at time $t-1$. The covariance matrix can be interpreted as the risk associated with the assets in our portfolio. According to the Modern Portfolio Theory introduced by Markowitz in his seminal paper [15], a risk-neutral investor should solve the following portfolio optimization problem to maximize his expected returns while minimizing his risk:

$$\max_{\omega_\mathbf{t} \in \mathrm{R^n}} \alpha_t^T \omega_\mathbf{t} - \frac{\lambda}{2} \omega_\mathbf{t}{}^T \Sigma_t \omega_\mathbf{t} \Rightarrow \omega_t^* = \frac{1}{\lambda} \Sigma_t^{-1} \alpha_t \tag{3}$$

This problem, called the *mean-variance optimization (MVO)* problem, although having the advantage of having a very simple solution, works so poorly in practice that optimization is often abandoned and replaced by suboptimal portfolios such as the equally-weighted portfolio [16]. In this paper, we use the solution proposed by [16] to solve this problem. The first step towards solving this problem is identifying the problem portfolios. By writing $\forall t (1, ..., T), \Sigma_t = \sigma_t \Omega_t \sigma_t$, $\Omega_t$ being the correlation matrix of the assets and $\sigma_t = \mathrm{diag}(\sigma_1, ..., \sigma_n)_t$ being the asset volatilities. $\Omega_t$ being a real symmetric matrix, it can be written as: $\Omega_t = P_t D_t P_t^T$, where $P_t \in \mathcal{O}_n(\mathrm{R}), D = \mathrm{diag}(D_1, ..., D_n)_t$. We can now write the initial portfolio optimization problem as:

$$\max_{\omega_t \in \mathrm{R}^n} (P_t^T \sigma_t \omega_t)^T P_t^T \sigma_t^{-1} \alpha_t - \frac{\lambda}{2} (P_t^T \sigma_t \omega_t)^T D (P_t^T \sigma_t \omega_t) \tag{4}$$

$$\Rightarrow \forall i (1, ..., n),\ \zeta_t^{i^*} = (P_t^T \sigma_t \omega_t)_i^* = \frac{1}{\lambda} \frac{(P_t^T \sigma_t^{-1} \alpha_t)_i}{\sqrt{D_{it}}} . \frac{1}{\sqrt{D_{it}}} \tag{5}$$

The portfolios $P_t^T \sigma_t^{-1}$ are defined as the eigen-vector portfolios of the covariance matrix and are called the *Principal Component* portfolios in the literature. The least important principal components of the covariance matrix are those with the lowest volatilities $\sqrt{D_i}$. Any error in the estimation of risk will likely lead to an underestimation of the risk of these portfolios because they have been chosen as the lowest-risk portfolios and any noise in the estimation of the expected return will likely be large relative to their risk. Therefore, a solution to this problem is to shrink the correlation matrix: $\tilde{D}_t = (1 - \theta) D_t + \theta I_n$ where $\theta \in [0, 1]$ is the shrinkage coefficient. This leads to the correlation matrix $\tilde{\Omega}_t = P \tilde{D}_t P^T = (1 - \theta) \Omega_t + \theta I_n$ hence the covariance matrix is adjusted as $\tilde{\Sigma}_t = \sigma_t \tilde{\Omega}_t \sigma_t$.

The last part of our portfolio construction methodology involves estimating the inverse of the covariance matrix of the returns $\mathbf{\Sigma}_t = \mathbf{V}(\mathbf{r}_{t+1} | \mathcal{F}_t)$. We will use an exponentially weighted covariance matrix. For a given $\rho \in (0, 1)$:

$$\hat{\mathbf{\Sigma}}_t = (1 - \rho) \sum_{k=0}^{t} \rho^k \mathbf{r}_{t-k} \mathbf{r}_{t-k}^T \tag{6}$$

The idea is to localize the data in time. As the spectral properties of the covariance matrix may be affected by our sample estimation, we use Graphical LASSO [17] which uses the $L_1$ penalty to estimate the precision matrix of the returns based on $\hat{\mathbf{\Sigma}}_t$.

### 5.2 Sign of return forecast

In a classification setting, the algorithms described above are designed to provide us with a forecast of the sign of the next-day returns of the stocks in our universe. In this case, we adopt the following portfolio construction methodology, where $n_+$ (resp. $n_-$) is the number of stocks with positive (resp. negative) returns:

$$\forall i \in (1, ..., n),\ w_{i,t} = \begin{cases} \frac{1}{n_+} \text{ if } \widehat{sgn(r_{i,t+1})} = 1, \\ -\frac{1}{n_-} \text{ if } \widehat{sgn(r_{i,t+1})} = -1 \end{cases} \tag{7}$$

In our classification problem, we consider the transformation $x \rightarrow \frac{x+1}{2}$ and its reciprocal $x \rightarrow 2x - 1$ to classify the sign of the returns as a 0/1 classification problem and then map it back to the $-1/1$ interval which justifies the use of the **categorical cross-entropy loss**.

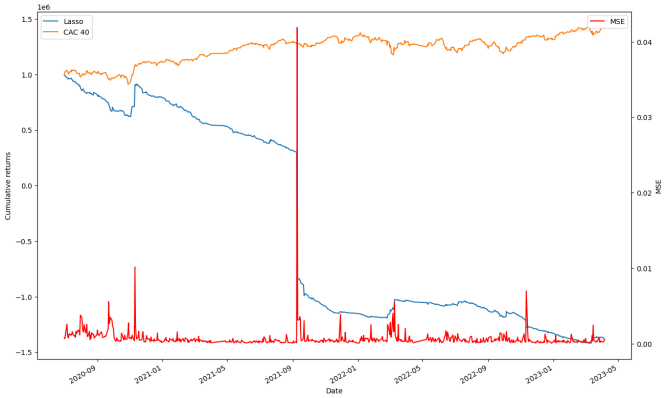
*Note:* We assume that our portfolio is fully allocated at all time, and that we can invest in the risk-free rate. For our strategies to be implemented in practice, we would need to account for transactions costs and market impact in our portfolio optimization problem. For simplicity, however, we assume fixed transaction costs of 0.33% based on the study by [18].

## 6 Results

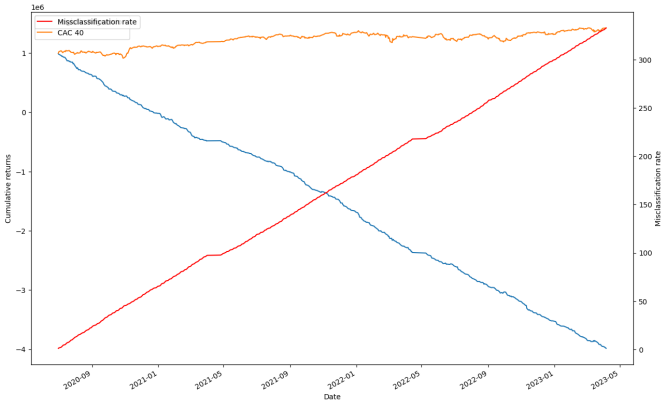
For all the models considered, we apply time-series validation to evaluate their performances. We need to train our models on a set of data that is older than the test data. There are two major approaches: the sliding window approach and the expanding window approach [19]. Under the expanding window strategy, the training split $L_t$ expands as the origin moves forward in time while the validation ($L_v$) split size remains constant. In other words, under the expanding window strategy, we choose all the data that is available before the origin as the training split. This effectively increases the training length every time the origin moves forward in time. In the rolling window strategy, we keep the length of the training split constant. The former approach is particularly useful if there is a limited amount of data to work with as in our case. Therefore, we use expanding window time-series validation. Based on [2], the performance metrics used to compare our models should be risk-adjusted returns based metrics, such as the Sharpe ratio, the Sortino ratio and the Calmar ratio. The Sharpe ratio, Sortino ratio and Calmar ratio are all defined as the ratio between the excess returns of the strategy and a measure of risk. For the Sharpe ratio, the risk is the standard deviation of the returns, for Sortino it is the standard deviation of the negative returns and for the Calmar it is the maximum drawdown of the returns. That is, for a given batch $((X_{train}, y_{train}), (X_{validation}, y_{validation}), (X_{test}, y_{test}))$, we build our predictor on the train set. On the validation set, we then determine the optimal set of hyper-parameters for the model and portfolio optimization. The criteria used is the simple average of Sharpe, Sortino and Calmar ratios. We then build the weights of the portfolio on the test set, and compare the performance of the various algorithms on the latter using an expanding window for the train set and fixed sizes (66 business days which correspond to 3 months) for the validation and test sets.

### 6.1 Benchmark models: Linear and logistic regression

As a benchmark, we include the linear and logistic regression models and compare their performance against the CAC40 index. For linear and logistic regression, we use a Lasso penalty to regularize our parameters.




(a) Lasso




(b) Logistic regression

We plot the cumulative returns of our strategies and the cumulative returns of the index on the primary y-axis. On the secondary y-axis, for linear regression, we plot the mean-squared error. For logistic regression, we plot the cumulative misclassification rate. The overall performance statistics of the benchmark models are reported in Table 1.

Table 1: Summary statistics for baseline linear models

| **Model** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Total mean-squared error** |
|---|---|---|---|---|
| LASSO | -92.2% | 72.7% | -1.27 | 0.46 |
| **Model** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Misclassification rate** |
| Logistic Reg | -182.3% | 13.4% | -13.6 | 50.4% |

We can see that the start of the Ukraine conflict in September 2021 lead to a drastic performance decrease for linear regression. The performance decrease can be explained by looking at the mean-squared error on the secondary y-axis. This same period was marked by a spike in mean-squared error due to our model taking to long to adjust as new patterns statistical emerged on the market.
For logistic regression, on the other hand, the strategy steadily looses capital. This can be explained by the steadily increasing cumulative misclassification rate on the secondary y-axis. The average accuracy of logistic regression during the test period is of $50.4\% \sim 50\%$, worst than picking the signs of the returns of the stocks at random. **These observations justify the need for better techniques to accomplish both forecasting tasks.** In the rest of this section, we use the **mean-squared error loss** for regression and the **categorical cross-entropy loss** for classification, which are defined as:

$$\forall y, \hat{y} \in \mathrm{R}^n, MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{8}$$

$$\forall y, \hat{y} \in [0,1]^n, CE(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n} log(\hat{y}_i^{y_i}(1-\hat{y}_i)^{(1-y_i)}) \tag{9}$$

## 6.2 Multi-layer perceptron

The Multi-Layer Perceptron model was constructed using linear hidden layers with the ReLU activation. Varying architecture hyperparameters for both regression and classification were compared (as summarized in Table 2). Specifically, we compare the results of varying the depth and width of the model for both regression and classification. We can see that the best performing model in a regression setting is the MLP with 10 layers of dimension 32 while the best model in a classification setting in the MLP with 50 layers of dimension 64. The obtained models tend to under-perform the market index but outperform the linear models in terms of Sharpe ratio. Further, in a regression setting, the MSE of the best performing MLP is much higher than that of LASSO. In a classification setting, the average daily misclassification error of the best performing MLP is lower than that of logistic regression and this model yields positive yearly returns but fails to outperform the benchmark market index in terms of Sharpe ratio.

Table 2: Summary statistics for MLP models

| **Regression Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Total mean-squared error** |
|---|---|---|---|---|
| **10 layers of dimension 32** | **-13.14%** | **204.2%** | **-0.06** | **11.74** |
| 10 layers of dimension 64 | -20.5% | 207.3% | -0.09 | 11.43 |
| 50 layers of dimension 64 | -2.9% | 3.3% | -0.87 | 0.28 |
| **Classification Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Misclassification Error** |
| 10 layers of dimension 32 | -198.3% | 19.5% | -10.15 | 0.51 |
| 10 layers of dimension 64 | -167.6% | 17.0% | -9.85 | 0.51 |
| **50 layers of dimension 64** | **1.6%** | **12.1%** | **0.13** | **0.49** |

## 6.3 Convolutional neural network

For the convolutional neural network, we compare models with varying depths and channel numbers as summarized in Table 3. As seen in the table, the model with 2 layers of [32,64] channels performed best for regression, and the model with 1 layer of 128 channels performed best in classification. The best performing regresssion model has high positive yearly returns and a Sharpe ratio higher than both the baseline and the CAC40. However, the volatility and mean-squared error of the CNN is much higher. The best performing classification model has higher positive returns, a higher Sharpe ratio, and a lower misclassification rate compared to the baseline logistic regression. It also outperforms the CAC40 with respect to returns and Sharpe ratio.

Table 3: Summary statistics for CNN models

| **Regression Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Total mean-squared error** |
|---|---|---|---|---|
| 1 layer of 64 channels | 123.1% | 265.3% | 0.46 | 27.2 |
| 1 layer of 128 channels | -247.4% | 564.3% | 0.43 | 33.4 |
| **2 layers of 32, 64 channels** | **404.1%** | **414.5%** | **0.97** | **25.9** |
| 3 layers of 32, 64, 128 channels | 243.1% | 302.1% | 0.80 | 29.0 |
| **Classification Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Misclassification Error** |
| 1 layer of 64 channels | -7.1% | 17.2% | -0.41 | 0.48 |
| **1 layer of 128 channels** | **64.4%** | **66.1%** | **0.97** | **0.47** |
| 2 layers of 32, 64 channels | -397.7% | 33.4% | -11.9 | 0.54 |
| 3 layers of 32, 64, 128 channels | -147.9% | 20.2% | -7.33 | 0.52 |

## 6.4 Recurrent neural network: LSTM

The LSTM model was first initialized as a network with 1 hidden layer and 1 final linear activation to observe its baseline performance. We then compare this to a model with 2 LSTM layers, and the results of these models can be seen in Table 4 for regression and classification. As seen in the table, the 1 layer model financially outperformed the 2 layer model for both regression and classification, as its yearly Sharpe ratio is higher. LSTM regression also produced cumulative returns higher than those of the CAC40; however, the high volatility results in a worse Sharpe ratio. Both classification models exhibited a high misclassification error of above 0.5, and both regression models also have a much higher mean-squared error compared to the baseline.

Table 4: Summary statistics for LSTM models

| **Regression Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Total mean-squared error** |
|---|---|---|---|---|
| **1 layer** | **60.6%** | **188.4%** | **0.32** | **12.78** |
| 2 layers | 32.8% | 140.8% | 0.23 | 9.95 |
| **Classification Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Misclassification Error** |
| **1 layer** | **-376.3%** | **14.7%** | **-25.5** | **0.53** |
| 2 layers | -412.9% | 14.4% | -28.7 | 0.52 |

## 6.5 Recurrent neural network: GRU

For the Gated Recurrent Model, we examined networks containing 1 GRU layer and 2 GRU layers, as seen in Table 5. The 1 layer model outperforms the 2 layer model in both regression and classification tasks with respect to the Sharpe ratio; however, this model also exhibits higher volatility, mean-squared error, and misclassification error. Furthermore, all of these models underperform the CAC40, and the classification models underperform the benchmark logistic regression. The regresssion models outperform the Lasso benchmark financially (higher Sharpe ratio), but perform much worse statistically and have much higher mean-squared errors.

Table 5: Summary statistics for GRU models

| **Regression Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Total mean-squared error** |
|---|---|---|---|---|
| **1 layer** | **-37.5%** | **125.7%** | **-0.29** | **13.33** |
| 2 layers | -95.6% | 76.5% | -1.25 | 10.42 |
| **Classification Models** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Misclassification Error** |
| **1 layer** | **-423.7%** | **17.2%** | **-24.6** | **0.55** |
| 2 layers | -399.9% | 14.4% | -27.8 | 0.49 |

## 6.6 Transformer: TFT

For Time-Fusion Transformer, we use the *Optuna* [20] framework to determine, for each stock, its set of optimal hyper-parameters on the first training and validation sets. We then keep this set of hyper-parameters **constant** as we move forward with our time-series validation procedure. **Although sub-optimal, we are only validating the portfolio optimization parameters at each iteration.** The reason for this is **computational complexity**: it is too computationally expensive to determine the optimal set of hyper-parameters for each iteration and each stock.

Table 6: Summary statistics for TFT models

| | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Total mean-squared error** |
|---|---|---|---|---|
| TFT (regression) | -32.3% | 10% | -3.21 | 0.39 |
| | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Misclassification Error** |
| TFT (classification) | 0.5% | 9.4% | 0.0055 | 49.1% |

We can see from Table 6 above that the time-fusion transformer drastically outperforms the linear regression model. In fact, the maximum mean-squared error loss of the transformer in the regression setting is around 2%, which is a 50% decrease when compared to Lasso regression. Moreover, in a classification setting, the time-fusion transformer strategy achieves positive yearly returns of 0.5% with a volatility of 9.3%, clearly over-performing logistic regression whose annual returns are of -182%. On average, moreover, its misclassification rate is of 49.1% which means that the model has an actual predictive ability and is not merely random guessing. However, the Time-Fusion Transformer fails to outperform the market index.

## 6.7 Summary

To summarize, we give the performance of the best performing model for each model type (MLP, CNN, LSTM, GRU, TFT) as well as the performance of the market index (CAC40) during the same period. The overall best model is the CNN for both classification and regression tasks as can be seen in Table 8 and Table 9. Our overall best model, the CNN model with 1 Conv1D layer of 128 channels, which has both very high Sharpe ratio and very low misclassification rate, drastically outperforms the market index based on Table 10.

Table 7: Summary statistics - benchmark

| | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** |
|---|---|---|---|
| CAC40 | 13.2% | 18.9% | 0.7 |

Table 8: Summary statistics - regression

| **Model** | **Yearly returns** | **Yearly volatility** | **Yearly Sharpe ratio** | **Total mean-squared error** |
|---|---|---|---|---|
| LASSO | -92.2% | 72.7% | -1.27 | 0.46 |
| MLP | -2.9% | 3.3% | -0.87 | 0.28 |
| **CNN** | **404.1%** | **414.5%** | **0.97** | **25.94** |
| LSTM | 32.8% | 140.8% | 0.23 | 9.95 |
| GRU | -37.5% | 125.7% | -0.29 | 13.33 |
| TFT | -32.3% | 10.0% | -3.2 | 0.39 |

Table 9: Summary statistics - classification

| Model | Yearly returns | Yearly volatility | Yearly Sharpe ratio | Misclassification rate |
|---|---|---|---|---|
| Logistic Reg | -182.3% | 13.4% | -13.6 | 50.4% |
| MLP | 1.6% | 12.1% | -0.13 | 48.5% |
| **CNN** | **64.4%** | **66.1%** | **0.97** | **47.2%** |
| LSTM | -376.3% | 14.7% | -25.5 | 52.8% |
| GRU | -399.9% | 14.4% | -27.8 | 49.0% |
| TFT | 0.5% | 9.3% | 0.0055 | 49.1% |

Table 10: Summary statistics - best performing portfolio vs. benchmark

| | Yearly returns | Yearly volatility | Yearly Sharpe ratio |
|---|---|---|---|
| CAC40 | 13.2% | 18.9% | 0.7 |
| **CNN** | **64.4%** | **66.1%** | **0.97** |

# 7 Further directions

The main improvement areas of our paper are as follows:

- To avoid predicting noise in the sign of the returns, we could perform a three class classification, assigning a label of zero to the returns that are not significantly different from zero.
- Due to limited computational capabilities, the scope of the cross-validated parameters was relatively narrow. Increasing the range of cross-validated parameters can lead to better results overall.
- Our models could be explored in the context of different markets and financial data-sets to observe their performance truly out-of-sample.
- More advanced models which combine different architectures (CNN + LSTM, CNN + GRU, GRU + LSTM...) were not included in our study and could lead to better performance overall.
- Further research could be conducted regarding our covariance matrix estimation methodology.

# 8 Conclusion

In conclusion, we found that, in a regression setting, none of the deep learning models examined were able to outperform the market index while having acceptable statistical performance metrics as compared to the baseline linear regression model. While the MLP and TFT models were able to statistically outperform standard linear models for predicting returns, they under-performed the market index. The CNN model, in contrast, was able to outperform the market index, but exhibited poor statistical performance with high mean-squared error. In a classification setting, all the models except LSTM were able to outperform the baseline logistic regression. However, only the CNN was able to outperform the market index. Further, we can see that the relationship between statistical performance (misclassification rate) and Sharpe ratio is not clear. For example, the MLP has lower misclassifcation rate than the TFT but underperforms the latter in terms of Sharpe ratio. **Can deep learning techniques effectively forecast future returns from past information?** Deep learning models for regression can be used to predict market returns but we could not find a regression model which has an acceptable mean-squared error while outperforming the market index. In terms of classification, they significantly outperform standard linear models when it comes to predicting the sign of those returns. The best convolutional neural network achieves the highest Sharpe ratio with the lowest misclassification rate. **How can we build, based on those forecast, a tradable portfolio? Can this portfolio outperform the market?** Using portfolio optimization to build our portfolio leads to good performance overall as we were able to outperform the CAC40 benchmark based on our portfolio construction methodologies. Our best performing model yields a Sharpe ratio of 0.97 while the benchmark CAC40 has a Sharpe ratio of 0.7 over the same period. **How can we evaluate the quality of a given model?** Statistical and risk-return performance metrics are not always aligned hence they should both be considered when assessing the portfolio performance.

# References


[1] Huang, J., Chai, J., Cho, S. Deep learning in finance and banking: A literature review and classification, 2020

[2] Jean Dessain, Machine learning models predicting returns: Why most popular performance metrics are misleading and proposal for an efficient metric, Expert Systems with Applications, Volume 199, 2022

[3] Hum Nath Bhandari, Binod Rimal, Nawa Raj Pokhrel, Ramchandra Rimal, Keshab R. Dahal, Rajendra K.C. Khatri, Predicting stock market index using LSTM, Machine Learning with Applications, Volume 9, 2022

[4] Sarker, I.H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. SN COMPUT. SCI. 2, 420, 2021

[5] Cybenko, G., Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems, 2(4), 303–314, 1989

[6] Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press. p. 326, 2016

[7] Serkan Kiranyaz and Onur Avci and Osama Abdeljaber and Turker Ince and Moncef Gabbouj and Daniel J. Inman, 1D Convolutional Neural Networks and Applications: A Survey, 2019

[8] Ashish Vaswani and Noam Shazeer and Niki Parmar and Jakob Uszkoreit and Llion Jones and Aidan N. Gomez and Lukasz Kaiser and Illia Polosukhin, Attention Is All You Need, 2017

[9] George Bird and Maxim E. Polivoda, Backpropagation Through Time For Networks With Long-Term Dependencies, 2021

[10] Bengio, Simard and Frasconi, Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks, 5(2), 157-166, 1994

[11] Hochreiter and Schmidhuber, Long short-term memory, Neural computation, 9(8), 1735–1780, 1997

[12] Cho, Van Merriënboer, Bahdanau and Bengio. On the properties of neural machine translation: encoder-decoder approaches, 2014

[13] Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey E. Hinton, Layer Normalization, 2016

[14] Bryan Lim, Sercan O. Arik, Nicolas Loeff and Tomas Pfister, Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting, 2020

[15] Harry Markowitz, Portfolio Selection, The Journal of Finance, Vol. 7, No. 1., pp. 77-91, 1952

[16] Lasse Heje Pedersen, Abhilash Babu, and Ari Levine, Enhanced Portfolio Optimization, Financial Analysts Journal, 77(2): 124-151, 2021

[17] Jerome Friedman, Trevor Hastie and Robert Tibshirani, Sparse inverse covariance estimation with the graphical lasso, 2007

[18] Hendrik Bessembinder, Trade Execution Costs and Market Quality after Decimalization, The Journal of Financial and Quantitative Analysis, Vol. 38, No. 4, pp. 747-777 (31 pages), 2003

[19] Bell, F., Smyl, S., 2018. Forecasting at Uber: An introduction. Accessed on 2023-04-23. `https://eng.uber.com/forecasting-introduction/`

[20] `https://optuna.org/`

[21] Wei Bao, Jun Yue and Yulei Rao, A deep learning framework for financial time series using stacked autoencoders and long-short term memory, 2017

[22] Jingyi Shen and M. Omair Shafiq, Short-term stock market price trend prediction using a comprehensive deep learning system, 2020

[23] Ryo Akita, Akira Yoshihara , Takashi Matsubara, Kuniaki Uehara, Deep learning for stock prediction using numerical and textual information, 2016

[24] M. Nabipour, P. Nayyeri, H. Jabani, A. Mosavi, E. Salwana, and S. S., "Deep Learning for Stock Market Prediction," Entropy (Basel), vol. 22, no. 8, p. 840, Jul. 2020, doi: 10.3390/e22080840.

[25] Mukherjee, S., et al.: Stock market prediction using deep learning algorithms. CAAI Trans. Intell. Technol. 8( 1), 82– 94 (2023). https://doi.org/10.1049/cit2.12059

[26] B L, S. and B R, S. (2023), "Combined deep learning classifiers for stock market prediction: integrating stock price and news sentiments", Kybernetes, Vol. 52 No. 3, pp. 748-773. https://doi-org.ezproxy.princeton.edu/10.1108/K-06-2021-0457

[27] K. Rekha and M. Sabu, "A cooperative deep learning model for stock market prediction using deep autoencoder and sentiment analysis," PeerJ Comput Sci, vol. 8, p. e1158, Nov. 2022, doi: 10.7717/peerj-cs.1158.

[28] Jimmy Lei Ba and Jamie Ryan Kiros and Geoffrey E. Hinton, Layer Normalization, 2016