National research university
«Higher School of Economics»
Faculty of Economic Siencies

TERM PAPER

An Application of Deep Learning to Algorithmic Trading for Time Series Forecasting Using Deterministic and Metaheuristic Optimization

Written by:
Gleb Khaykin
Group BEC1812

Advisor:
Petr Lukyanchenko
ICEF, FCS Lecturer

Moscow — 2021

Contents

1. Introduction

Algorithmic trading is a process for executing buy and sell orders using an automated algorithm that follows a defined set of instructions based on timing, price, quantity, and other criteria to place a trade for a financial asset. It is closely related to time series forecasting, for the goal of quantitative researchers, or quants, is to recognize the trend in the analyzed time series based on the asset's price movements to generate trading signals.

Conventional time series forecasting models have been widely used to construct predictions, such as the ARIMA models. Nevertheless, due to a statistical foundation and therefore its constraints, such models can perform quite poorly on financial data since financial markets are regarded as non-linear dynamic systems.

So, we want to introduce recurrent neural networks (RNNs) that can learn complex dimensionality of the financial time series, which is essential to improving prediction performance. However, neural networks have a problem choosing parameters for the model, and solely the user's experience mostly determines them. To cope with this problem, metaheuristic optimization is considered such as particle swarm optimization [1] and [2], genetic algorithm [3], and so forth.

The purpose of this paper is to construct GA-optimized RNNs, namely LSTM and GRU networks, and compare their performance to the ARIMA model's forecast used as a benchmark. In addition, we want to test whether technical indicators are helpful for financial time series forecasting, as a RNN architecture allows us to use a multivariate time series.

The structure of the paper is organized as follows. The theories of ARIMA, LSTM, GRU, deterministic optimization, and metaheuristic optimization are explained in section 2. Then, in section 3, the covered theory is implemented in practice by proposing ARIMA, GA-LSTM, GA-LSTM-TI, GA-GRU, GA-GRU-TI models. Finally, in section 4, a comparison of the forecasts is presented with the consequent conclusion.

2. Theory

2.1. Autoregressive Integrated Moving Average Model

As it has been already noted, an autoregressive integrated moving average, or ARIMA, model is the most widely used approach to time series forecasting. It attempts to capture the linear relationship between response variable y_t and its lagged values $y_{t-1}, \ldots, y_{t-k}$, that is information available earlier. However, before introducing this model, we must first define the notion of time series, stationarity, and other concepts.

2.1.1. Stationarity

The stationarity condition is critically important for time series analysis, for many statistical models rely on it. We regard time series as a sequence of random variables $\{y_t\}_{t=1}^{T}$ over time given one probability space.

A series is said to be strongly stationary if $F(y_{t_1}, \ldots, y_{t_m}) = F(y_{t_1+k}, \ldots, y_{t_m+k})$ $\forall k, t_1, \ldots, t_m \in \mathbb{R}, m \in \mathbb{N}$. In other words, the joint distribution of $\{y_{t_1}, \ldots, y_{t_m}\}$ does not depend on the time step t. It means that time series with trends or seasonality are not considered to be stationary since their statistical properties are independent of the time step t.

However, strong stationarity is a condition that is hard to verify in practice. Hence, weak stationarity is often considered. A series $\{y_t\}_{t=1}^{T}$ is said to be weakly stationary if it has time-invariant first and second moments $\forall t, k \in \{1, \ldots, T\}$:

$$\begin{aligned} \mathbb{E}[y_t] &= \mu \\ \mathbb{V}ar[y_t] &= \sigma_\varepsilon^2 \\ \mathbb{C}ov[y_t, y_{t-k}] &= \gamma_k \end{aligned}$$

A simple example of a stationary process is a white noise $\{\varepsilon_t\}_{t=1}^{T}$, where $\varepsilon_t \sim \mathrm{WN}(0, \sigma_\varepsilon^2)$. One considers a process as a white noise if it represents a sequence of independent and identically distributed random variables from a fixed distribution with zero mean, $\mathbb{E}[\varepsilon_t] = 0$, and constant variance, $\mathbb{V}ar[\varepsilon_t] = \sigma_\varepsilon^2$. Moreover, if we assume that a distribution is normal, that is $\varepsilon_t \sim \mathcal{N}(0, \sigma_\varepsilon^2)$, the series is called a Gaussian white noise.

2.1.2. Autoregressive and Moving Average Models

Wold Representation Theorem

The Wold's theorem [4] states that if y_t is a weakly stationary process, then it can be represented as

$$y_t = \eta_t + \sum_{i=0}^{\infty} \psi_i \varepsilon_{t-i},$$

where η_t is a deterministic time series, $\{\psi_i\}_{i=0}^{\infty}$ are weights such that $\sum_{i=0}^{\infty} \psi_i < \infty$, and ε_t is a white noise process.

We can obtain its mean and variance using independence of ε_t:

$$\begin{aligned} \mathbb{E}[y_t] &= \eta_t \\ \mathbb{V}ar[y_t] &= \sigma_\varepsilon^2 \sum_{i=0}^{\infty} \psi_i^2 \end{aligned}$$

The usefulness of the Wold's decomposition is that it allows for a variable y_t to be approximated by a linear model. And therefore, we can introduce an alternative model with a finite number of parameters that would be equivalent to the infinite sum of the white noise process.

Moving Average Model

A moving average model of order q, MA(q), is the same as a multiple linear regression where its regressors are the current and lagged values of a stochastic term and written as

$$y_t = \sum_{i=1}^{q} \psi_i \varepsilon_{t-i} + \varepsilon_t, \ \varepsilon_t \sim \mathrm{WN}(0,\sigma_\varepsilon^2)$$

where $\{\psi_i\}_{i=1}^{q}$ are weights and $\{\varepsilon_{t-i}\}_{i=0}^{q}$ are white noise error terms.

It is obvious that MA(q) model is always stationary, since

$$\begin{aligned} \mathbb{E}[y_t] &= 0 \\ \mathbb{V}ar[y_t] &= \sigma_\varepsilon^2 \left(\sum_{i=1}^{q} \psi_i^2 + 1 \right) \end{aligned}$$

Autoregressive Model

An autoregressive model of order p, AR(p), is the same as a multiple linear regression where its regressors are lagged values of the same series and written as

$$y_t = \alpha + \sum_{i=1}^{p} \varphi_i y_{t-i} + \varepsilon_t,\ \varepsilon_t \sim \text{WN}(0,\sigma_\varepsilon^2)$$

where α is constant, $\{\varphi_i\}_{i=1}^{p}$are weights, and ε_t is a white noise error term.

The AR(p) model can be rewritten using the notation of the lag operator, $L^i y_t = y_{t-i}$, as

$$\left(1 - \sum_{i=1}^{p} \varphi_i L^i\right) y_t = \alpha + \varepsilon_t$$

$$\varphi_p(L) y_t = \alpha + \varepsilon_t, \varphi_p(L) = 1 - \varphi_1 L - \varphi_2 L^2 - \ldots - \varphi_p L^p$$

The operator $\varphi_p(L)$ can be interpreted as a filter that when applied to the series y_t it converts to a white noise process ε_t:

$$y_t = \frac{1}{\varphi_p(L)}(\alpha + \varepsilon_t)$$

We can show that the AR(p) process is stationary and has a pure MA representation, accordingly to the Wold's theorem, provided that the roots of the polynomial $\varphi_p(z)$ lie outside the unit circle, i.e. $\varphi(z) \neq 0\ \forall z \leqslant 1$.

To depict this, let us consider a simple AR(2) process:

$$y_t = \alpha + \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + \varepsilon_t$$

$$\left(1 - \varphi_1 L - \varphi_2 L^2\right) y_t = \alpha + \varepsilon_t$$

$$(1 - \zeta_1 L)(1 - \zeta_2 L)\, y_t = \alpha + \varepsilon_t$$

Here, the series is time reversible, if $|\zeta_1| < 1$ and $|\zeta_2| < 1$, and than y_t is stationary.

Autoregressive Moving Average Model

An autoregressive moving average model, ARMA(p,q), is a linear model with p AR lags and q MA lags and written as

$$y_t = \alpha + \sum_{i=1}^{p} \varphi_i y_{t-i} + \sum_{i=1}^{q} \psi_i \varepsilon_{t-i} + \varepsilon_t$$

$$\left(1 - \sum_{i=1}^{p} \varphi_i L^i\right) y_t = \alpha + \left(1 + \sum_{i=1}^{q} \psi_i L^i\right) \varepsilon_t$$

$$\varphi_p(L)y_t = \alpha + \psi_q(L)\varepsilon_t$$

It can be purely seen that the stationarity of the ARMA(p,q) process is solely dependent on the AR(p) process.

Autoregressive Integrated Moving Average Model

If a series $\{y_t\}_{t=1}^T$ is assumed to be non-stationary, but a series $\{\Delta^d y_t\}_{t=1}^T$, on the contrary, is stationary, we use ARIMA(p,d,q) process for y_t with d-order of difference:

$$\Delta^d y_t = \alpha + \sum_{i=1}^{p} \varphi_i \Delta^d y_{t-i} + \sum_{i=1}^{q} \psi_i \varepsilon_{t-i} + \varepsilon_t$$

$$\left(1 - \sum_{i=1}^{p} \varphi_i L^i\right) \Delta^d y_t = \alpha + \left(1 + \sum_{i=1}^{q} \psi_i L^i\right) \varepsilon_t$$

$$\varphi_p(L)\Delta^d y_t = \alpha + \psi_q(L)\varepsilon_t$$

Let us assume the first-order differencing, $\Delta y_t = y_t - y_{t-1}$, then the point forecast for the ARIMA(p, 1, q) model is

$$\Delta\hat{y}_t = \hat{\alpha} + \hat{\varphi}_1 \Delta y_{t-1} + \cdots + \hat{\varphi}_p \Delta y_{t-p} + \varepsilon_t + \hat{\psi}_1 \varepsilon_{t-1} + \cdots + \hat{\psi}_q \varepsilon_{t-q}$$

$$\hat{y}_t = \hat{\alpha} + (\hat{\varphi}_1 + 1) y_{t-1} - (\hat{\varphi}_1 - \hat{\varphi}_2) y_{t-2} - \cdots - (\hat{\varphi}_{p-1} - \hat{\varphi}_p) y_{t-p} - \hat{\varphi}_p y_{t-p-1} + \\ + \varepsilon_t + \hat{\psi}_1 \varepsilon_{t-1} + \cdots + \hat{\psi}_q \varepsilon_{t-q}$$

and 95-% forcast interval is

$$\left(\hat{y}_t - 1.96\sqrt{\mathbb{V}ar(\hat{y}_t)}, \hat{y}_t + 1.96\sqrt{\mathbb{V}ar(\hat{y}_t)}\right)$$

Note that the second-order differencing is $\Delta^2 y_t = \Delta y_t - \Delta y_{t-1} = (y_t - y_{t-1}) - (y_{t-1} - y_{t-2}) = y_t - 2y_{t-1} + y_{t-2}$.

2.2. Feedforward Neural Network

Now, let us discuss more sophisticated forecasting methods based on the artificial neural networks (ANNs).

The simplest ANN is a feedforward network (FFNN) defined by the architecture $\hat{y}(x;w) = f_n(FC_n(f_{n-1}(FC_{n-1}(\dots f_1(FC_1(x))\dots))))$, where the ith fully connected layer FC_i is a function that takes as input $\{h_j\}_{j=1}^{d_{i-1}}$ neurons and outputs $\{h_j\}_{j=1}^{d_i}$ neurons, given that each output is a linear model over the inputs,

and the ith activation function $f_i : \mathbb{R} \to \mathbb{R}$ is a transformation function that decides what neurons of the ith fully connected layer FC_i is to be transfered to the next fully connected layer FC_{i+1}.

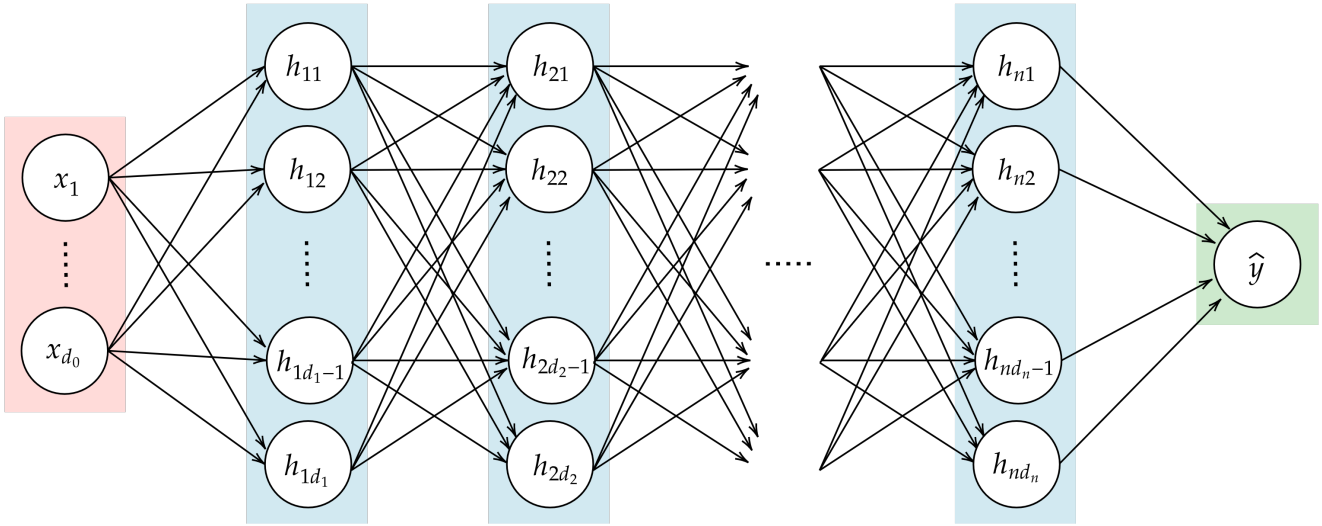

Figure 2.1 — Feedforward neural network architecture.

By noting the ith fully connected layer of the network and the jth hidden unit of the layer $\forall j \in \{1, \ldots, d_i\}$ and $\forall i \in \{1, \ldots, n\}$, we have

$$h_{ij}(x; w) = f_i\left(w_{ij}^T h_{i-1}(x; w) + b_{ij}\right)$$

where initially $h_{1j}(x; w) = w_{1j}^T x + b_{1j}$ $\forall j \in \{1, \ldots, d_0\}$.

The ith fully connected layer is represented as

$$h_i(x; w) = f_i\left(W_i h_{i-1}(x; w) + b_i\right)$$

2.2.1. Activation Functions

To understand the need for activation functions in the FFNN, consider two fully connected layers without a non-linear function between them:

$$\begin{aligned} h_i(x; w) &= W_i h_{i-1}(x; w) + b_i = W_i(W_{i-1} h_{i-2}(x; w) + b_{i-1}) + b_i \\ &= \underbrace{W_i W_{i-1}}_{new\ weight}\, h_{i-2}(x; w) + \underbrace{W_i b_{i-1} + b_i}_{new\ bias} \end{aligned}$$

As one can see, in such a case, two fully connected layers are equivalent to one fully connected layer. Thus, if we use only fully connected layers without activation functions, we will not be able to build anything more complicated than a linear

model. Hence, for this reason, non-linear functions are applied to the outputs of the layers.

Typical activation functions f_i used in practice are:

1. Rectified linear unit (ReLU)

$$\begin{aligned} \text{rect}(z) &= \max\{0,z\} \\ \frac{d}{dz}\text{rect}(z) &= \begin{cases} 1, & z > 0 \\ 0, & z \leqslant 0 \end{cases} \end{aligned}$$

2. Leaky ReLU

$$\begin{aligned} \text{leaky}(z) &= \max\{\alpha z, z\},\ \alpha \in (0,1) \\ \frac{d}{dz}\text{leaky}(z) &= \begin{cases} 1, & z > 0 \\ \alpha, & z \leqslant 0 \end{cases} \end{aligned}$$

3. Sigmoid

$$\begin{aligned} \sigma(z) &= \frac{1}{1+\exp(-z)} \\ \frac{d}{dz}\sigma(z) &= \frac{-\exp(-z)}{1+\exp(-z)} = \sigma(z)(1-\sigma(z)) \end{aligned}$$

4. Hyperbolic tangent

$$\begin{aligned} \tanh(z) &= \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)} = 2\sigma(2z) - 1 \\ \frac{d}{dz}\tanh(z) &= 1 - \left(\frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}\right)^2 = 1 - \tanh^2(z) \end{aligned}$$

2.2.2. Optimization

The optimization problem in deep learning is defined by introducing an algorithm that minimizes the objective function and written as

$$Q(\hat{y}, X) = \frac{1}{T}\sum_{t=1}^{T} L(y_t,\, \hat{y}(x_t; w)) \to \min_w,$$

where $X = \{(x_t, y_t)\}_{t=1}^{T}$ is a learning dataset and $\hat{y}(x_t; w)$ is an optimized model.

This subsection considers the most popular iterative first-order optimization algorithms, going from the simplest to the more complex ones.

Gradient Decent

Take into consideration the objective function $Q : \mathbb{R}^d \to \mathbb{R}$ that maps vectors into scalars. One knows that the gradient of the objective function $\nabla_w Q$ is a vector of partial derivatives

$$\nabla_w Q = \left(\frac{\partial Q}{\partial w_1}, \ldots, \frac{\partial Q}{\partial w_d} \right)$$

and the key property of a gradient is that it is the direction of the fastest growth of the function. Therefore, the anti-gradient is regarded as the steepest waning direction. Given that, we can start from some point $w^{(0)}$, and then shift towards the anti-gradient of the objective function:

$$w^{(k)} = w^{(k-1)} - \eta \nabla_w Q(w^{(k-1)})$$

where $w^{(k)}$ is a new point, $w^{(k-1)}$ is an old point, η is learning rate. After that we recalculate the anti-gradient of the objective function in the new point, and shift towards it again until reaching the stop criterion that can be written as $||\nabla_w Q(w^{(k)})||_2 < \varepsilon$. To put it another way, the GD stops when the training error on the data virtually stops decreasing.

Stochastic Gradient Descent

One knows that the objective function is $Q(\hat{y}, X) = \frac{1}{T} \sum_{t=1}^{T} L(y_t,\ \hat{y}(x_t; w))$, so the gradient of the objective function Q is the sum of the gradients of the loss function L given sample data, that is $\nabla_w Q(\hat{y}, X) = \frac{1}{T} \sum_{t=1}^{T} \nabla_w L(y_t,\ \hat{y}(x_t; w))$. It means that at every time step k of the GD it is necessary to calculate the gradient of the entire sum, and the computational cost for each independent variable iteration grows linearly with T. At the same time, the exact calculation of the gradient may not be that necessary, since we do not take very large steps towards the anti-gradient due to the use of the learning step η. Hence, the presence of inaccuracies in calculation of the gradient should not greatly affect the overall optimization trajectory, and we can use stochastic gradient descent (SGD) that reduces computational cost at each iteration by sampling a random learning object $(x_{t_i}, y_{t_i}) \in X$. Then computing the gradient of the loss function given that observation is written as

$$w^{(k)} = w^{(k-1)} - \eta \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}))$$

However, even when we approach the local minimum, the gradient for one observation is unlikely to be zero. Thus, there is a need to change the learning rate η such that it is epoch-dependent:

$$w^{(k)} = w^{(k-1)} - \eta_k \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}))$$

For example, the use of $\eta_k = \frac{1}{k^{0.3}}$ allows us to control convergence of an optimization algorithm.

Also, it is possible to increase the gradient estimate by using mini-batches $\mathcal{B}_k = \{(x_{t_i}, y_{t_i})\}_{i=1}^m$ which contain m observations instead of one:

$$w^{(k)} = w^{(k-1)} - \eta_k \frac{1}{|\mathcal{B}_k|} \sum_{(x_{t_i}, y_{t_i}) \in \mathcal{B}_k} \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}))$$

Momentum

One of the properties of the gradient is that it is orthogonal to the contour lines of the objective function. Hence, it may turn out that in the direction of the mini-batch SGD the anti-gradient varies greatly from time step to time step if the contour lines are strongly elongated. Because of the orthogonality of the gradient to the contour lines it will change direction virtually at every time step. Such oscillations will introduce a lot of noise into the movement of the mini-batch SGD, and the optimization will take more iterations. To avoid this, we use momentum which aggregates a history of past gradients to accelerate convergence:

$$h_k = \alpha h_{k-1} + \eta_k \frac{1}{|\mathcal{B}_k|} \sum_{(x_{t_i}, y_{t_i}) \in \mathcal{B}_k} \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}))$$

$$w^{(k)} = w^{(k-1)} - h_k$$

where h_k is average direction of the motion at time step k, which in the beginning is initialized at zero, $\alpha \in (0, 1)$ is a parameter that regulates how much strongly we want to take into account the past time steps.

If we want to speed up the convergence even more, we can use the Nesterov momentum [7]:

$$h_k = \alpha h_{k-1} + \eta_k \frac{1}{|\mathcal{B}_k|} \sum_{(x_{t_i}, y_{t_i}) \in \mathcal{B}_k} \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)} - \alpha h_{k-1})),$$

where $w^{(k-1)} - \alpha h_{k-1}$ is a fair estimate of the position where we get in the next time step.

AdaGrad and RMSProp

The mini-batch SGD is very sensitive to the choice of step length in the learning rate η. For instance, if the step is too immense, then there is a risk that we will jump over the local minimum. At the same time, if the step is too small, then many iterations are required to find the local minimum. There is no way to determine the correct step size in advance, and scheme η_k that gradually decreases the step can also work poorly.

So, in the AdaGrad method, proposed by Duchi et al. [8], there is an idea to make for each the jth component of the parameter vector its own step length. In this case, the operations are applied coordinate-wise, and the step will be the smaller, the longer steps we have made in the previous iterations:

$$g_j^{(k)} = g_j^{(k-1)} + \left(\frac{1}{|\mathcal{B}_k|} \sum_{(x_{t_i}, y_{t_i}) \in \mathcal{B}_k} \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}) \right)_j^2$$

$$w^{(k)} = w^{(k-1)} - \frac{\eta_k}{\sqrt{g_j^k + \varepsilon}} \left(\frac{1}{|\mathcal{B}_k|} \sum_{(x_{t_i}, y_{t_i}) \in \mathcal{B}_k} \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}) \right)_j$$

where ε is an additive constant that ensures that we do not divide by zero.

Nonetheless, the AdaGrad method has an issue. Step length can decrease too quickly because $g_j^{(k)}$ is monotonically increasings which leads to an early stop. For this reason, RMSProp is proposed by Tieleman and Hinton [9], solving this drawback by using exponential decay of gradients:

$$g_j^{(k)} = \alpha g_j^{(k-1)} + (1 - \alpha) \left(\frac{1}{|\mathcal{B}_k|} \sum_{(x_{t_i}, y_{t_i}) \in \mathcal{B}_k} \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}) \right)_j^2$$

Adam

Kingma and Ba [10] proposed an optimization algorithm called Adam which combines all the above-mentioned techniques:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) \left(\frac{1}{|\mathcal{B}_k|} \sum_{(x_{t_i}, y_{t_i}) \in \mathcal{B}_k} \nabla_w L(y_{t_i}, \hat{y}(x_{t_i}; w^{(k-1)}) \right)_j$$

$$\hat{m}_j^{(k)} = \frac{m_j^{(k)}}{1 - \beta_1^k}$$

$$g_j^{(k)} = \beta_2 g_j^{(k-1)} + (1-\beta_2)\left(\frac{1}{|\mathcal{B}_k|}\sum_{(x_{t_i},y_{t_i})\in\mathcal{B}_k}\nabla_w L(y_{t_i},\hat{y}(x_{t_i};w^{(k-1)})\right)_j^2$$

$$\hat{g}_j^{(k)} = \frac{g_j^{(k)}}{1-\beta_2^k}$$

$$w_j^{(k)} = w_j^{(k-1)} - \frac{\eta_k}{\sqrt{\hat{g}_j^{(k)}+\varepsilon}}\hat{m}_j^{(k)}$$

where β_1 and β_2 are non-negative parameters, $\hat{m}^{(k)}$ is momentum at the time step k, $\hat{g}^{(k)}$ is RMSProp-like part at the time step k, and ε is an additive constant that ensures that we do not divide by zero.

2.2.3. Backpropagation

A significant advantage of neural networks is that the output $\hat{y}(x;w)$ is differentiable by parameters. However, we would like to differentiate it effectively. Therefore, we use the backward propagation of errors concerning the neural network's weights using gradient decent methods described in the previous subsection. Backpropagation calculates the gradients in reverse order, from the output to the input layer, allowing us to use the chain rule of differentiation.

Firstly, let us calculate the derivatives of the loss function for the last layer $\hat{y}(x;w) = h_n(h_{n-1},w_n,b_n)$ using partial derivatives:

$$\frac{\partial L}{\partial w_n} = \frac{\partial L}{\partial h_n}\frac{\partial}{\partial w_n}h_n(h_{n-1},w_n,b_n)$$
$$\frac{\partial L}{\partial b_n} = \frac{\partial L}{\partial h_n}\frac{\partial}{\partial b_n}h_n(h_{n-1},w_n,b_n)$$
$$\frac{\partial L}{\partial h_{n-1}} = \frac{\partial L}{\partial h_n}\frac{\partial}{\partial h_{n-1}}h_n(h_{n-1},w_n,b_n)$$

Secondly, let us calculate the derivatives of the loss function for the penultimate layer $\hat{y}(x;w) = h_n(h_{n-1}(h_{n-2},w_{n-1},b_{n-1}),w_n,b_n)$:

$$\frac{\partial L}{\partial w_{n-1}} = \frac{\partial L}{\partial h_n}\frac{\partial h_n}{\partial h_{n-1}}\frac{\partial}{\partial w_{n-1}}h_{n-1}(h_{n-2},w_{n-1},b_{n-1}) =$$
$$= \frac{\partial L}{\partial h_{n-1}}\frac{\partial}{\partial w_{n-1}}h_{n-1}(h_{n-2},w_{n-1},b_{n-1})$$

$$\frac{\partial L}{\partial b_{n-1}} = \frac{\partial L}{\partial h_n}\frac{\partial h_n}{\partial h_{n-1}}\frac{\partial}{\partial b_{n-1}}h_{n-1}(h_{n-2}, w_{n-1}, b_{n-1}) =$$
$$= \frac{\partial L}{\partial h_{n-1}}\frac{\partial}{\partial b_{n-1}}h_{n-1}(h_{n-2}, w_{n-1}, b_{n-1})$$
$$\frac{\partial L}{\partial h_{n-2}} = \frac{\partial L}{\partial h_n}\frac{\partial h_n}{\partial h_{n-1}}\frac{\partial}{\partial h_{n-2}}h_{n-1}(h_{n-2}, w_{n-1}, b_{n-1}) =$$
$$= \frac{\partial L}{\partial h_{n-1}}\frac{\partial}{\partial h_{n-2}}h_{n-1}(h_{n-2}, w_{n-1}, b_{n-1})$$

Thus, considering all layers we get:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_i}\frac{\partial}{\partial w_i}h_i(h_{i-1}, w_i, b_i)$$
$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial h_i}\frac{\partial}{\partial b_i}h_i(h_{i-1}, w_i, b_i)$$
$$\frac{\partial L}{\partial h_{i-1}} = \frac{\partial L}{\partial h_i}\frac{\partial}{\partial h_{i-1}}h_i(h_{i-1}, w_i, b_i)$$

One can observe that computations of the gradient from one layer can be reused to compute the gradient for the previous layer. Hence, backpropagation stores any intermediate partial derivatives for calculations left. This algorithm is way more efficient than using a naive approach of calculating each layer's gradient separately.

2.3. Recurrent Neural Network

A recurrent neural network (RNN) is a type of the ANN that is well-suited to time series $\{y_t\}_{t=1}^T$, since it is designed to work with sequence data.

Let us assume that we want to maximize the probability of a series $\{y_t\}_{t=1}^T$. Then the model would be written as

$$p(y_1, \ldots, y_T) = \prod_{t=1}^{T} p(y_t \mid y_{t-1}, \ldots, y_1) \to \max$$

In order to get $\hat{y}_t$, there is a need to model the conditional probability given all previous values $\hat{y}_{t-1}, \ldots, \hat{y}_1$:

$$\hat{y}_t \sim p(y_t \mid \hat{y}_{t-1}, \ldots, \hat{y}_1)$$

Note that to get the probability of y_t we need to construct $t-1$ sequential models. In such a case, the memory requirements grow exponentially with the time step t.

However, instead of modeling this probability, we could use a latent variable h_t called a hidden state, which stores all data it has seen so far:

$$p(y_t \mid \hat{y}_{t-1}, \ldots, \hat{y}_1) \approx p(y_t \mid h_{t-1}),\ h_t = f(y_t, h_{t-1})$$

The RNN uses this technique. It is the ANN with hidden states and is capable of conditioning the model on all previous values. Consider a multivariate time series $\{(x_t, y_t)\}_{t=1}^{T}$, $x_t \in \mathbb{R}^d$ and $y_t \in \mathbb{R}$, then the RNN has the following architecture:

$$h_t = f_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$
$$\hat{y}_t = f_y(W_{hy}h_t + b_y)$$

where h_t is hidden state and $\hat{y}_t$ is output layer at the time step t, W_{xh}, W_{hh}, W_{hy} are weights, b_h, b_y are biases, and f_h, f_y are activation functions.

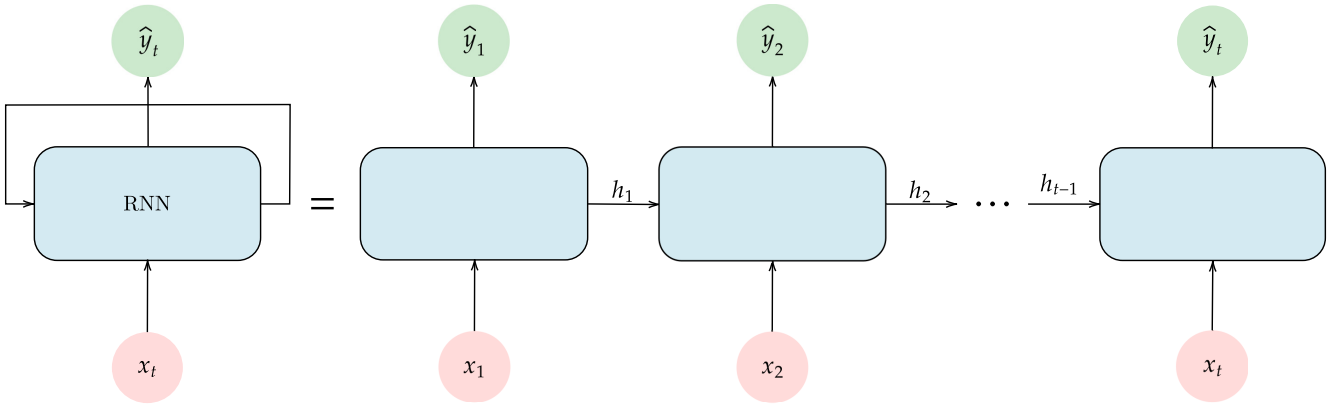

Figure 2.2 — Recurrent neural network architecture.

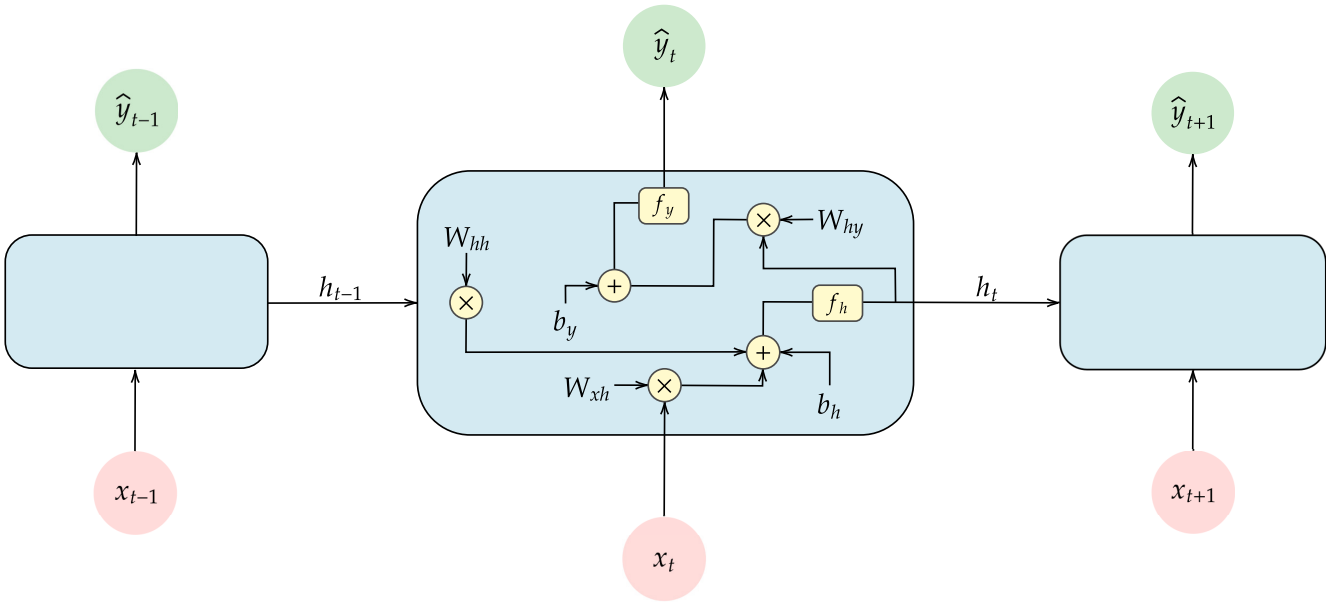

Figure 2.3 — Recurrent neural network block.

2.3.1. Backpropagation Through Time

To train a RNN model, we use backpropagation through time (BPTT), considering that now $L(y_t, \hat{y}(x_t; w)) = \sum_{t=1}^{T} L_t(y_t, \hat{y}_t(x_t; w))$.

The derivatives of the loss function when there are no recurrent loops are computed straightforward:

$$\frac{\partial L}{\partial b_y} = \sum_{i=t}^{T} \frac{\partial L_t}{\partial b_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial b_y}$$

$$\frac{\partial L}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial b_h} \frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial W_{hy}}$$

However, it gets a little bit more complicated when there are recurrent loops. Let $W \in \{W_{xh}, W_{hh}\}$, then the derivatives of the loss function are written as follows

$$\begin{aligned}
\frac{\partial L}{\partial W} &= \sum_{t=1}^{T} \frac{\partial L_t}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial W} = \\
&= \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \left(\frac{\partial h_t}{\partial W} + \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W} + \ldots + \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_1}{\partial W} \right) = \\
&= \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \sum_{i=1}^{t} \left(\prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_t}{\partial W}
\end{aligned}$$

Note that because a neuron $h \in \mathbb{R}^q$, each $\partial h_j / \partial h_{j-1}$ is the Jacobian matrix for h:

$$\frac{\partial h_j}{\partial h_{j-1}} = \left(\frac{\partial h_j}{\partial h_{j-1,1}}, \ldots, \frac{\partial h_j}{\partial h_{j-1,q}} \right) = \begin{pmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,1}}{\partial h_{j-1,q}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{j,q}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,q}}{\partial h_{j-1,q}} \end{pmatrix}$$

Now it can be inferred that the RNN has vanishing and exploding gradient problems:

1. If $\| \frac{\partial h_j}{\partial h_{j-1}} \|_2 < 1$, then the gradient is vanishing as the time step t is becoming large. It means that $W^{new} = W^{old} - \eta_t \frac{\partial L}{\partial W} \approx W^{old}$. To put it another way, our neural network stops learning.
2. If $\| \frac{\partial h_j}{\partial h_{j-1}} \|_2 > 1$, then the gradient is exploding as the time step t is becoming large. Enormous updates to the weight during training reaches a bad parameter configuration and sometimes can cause a numerical overflow, i.e. NaNs.

Given that, the RNN models often need help to stabilize their training. To solve the problem of exploding gradients, Pascanu et al. [11] introduced a simple solution that clips gradients to a small number whenever they explode:

$$g = \min\left(1, \frac{\delta}{\| g \|_2}\right) g$$

where δ is the threshold and $g = \frac{\partial L}{\partial W}$. Alas, we still have the vanishing gradient problem.

2.3.2. Long Short-Term Memory

The main problem of the RNN is that it struggles to preserve information over many time steps, since the hidden state is constantly being rewritten $h_t = f_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$. In order to cope with this, specific gates $\Gamma = f(W_x x_t + W_h h_{t-1} + b)$ are introduced, where W_x, W_h, b are weights specific to the gate and f is an activation function.

Hochreiter and Schmidhuber [12] proposed a RNN model with Long-Term Short-Term Memory (LSTM) blocks as a solution to the vanishing gradient problem. The LSTM block, depicted in figure 2.4, is designed to address long-term information preservation and short-term input skipping by using not only a hidden state h_t, but also a memory cell c_t which is able to remember information over time.

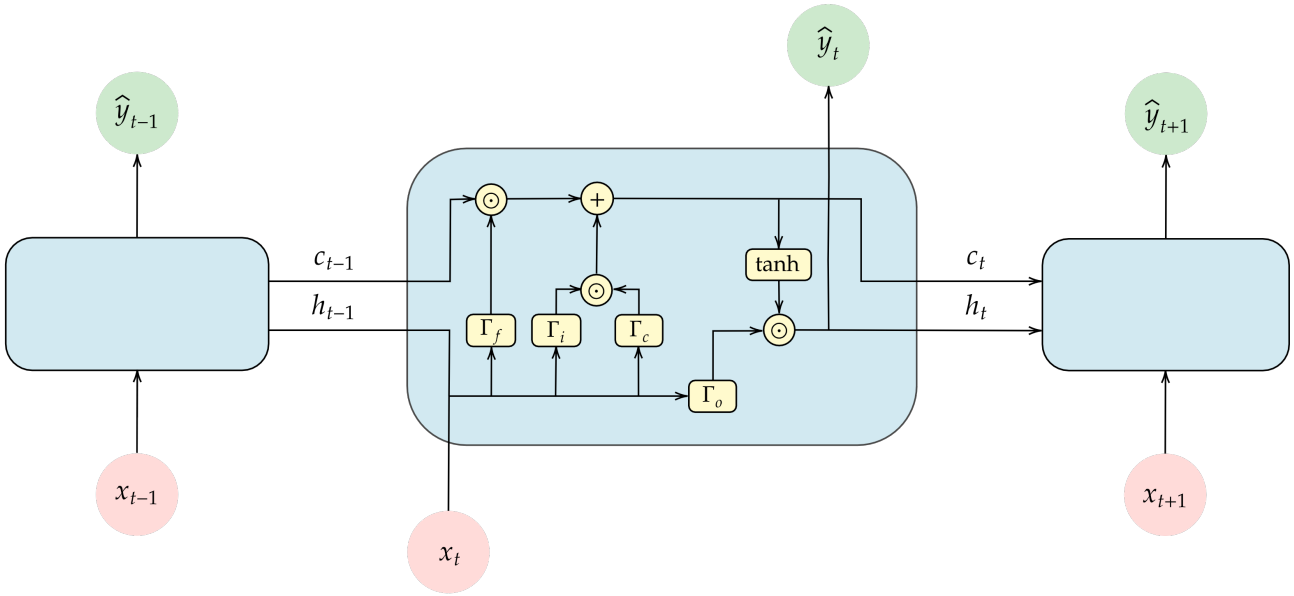

Figure 2.4 — Long short-term memory block.

It has the following architecture:

1. Hidden state h_t and cell state c_t:

$$c_t = \Gamma_i \odot \Gamma_c + \Gamma_f \odot c_{t-1}$$

$$h_t = \Gamma_o \odot \tanh(c_t)$$

2. Input gate that decides what parts of the new cell content are written to cell

$$\Gamma_i = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

3. Cell gate that decides what new content to be written to the cell

$$\Gamma_c = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

4. Forget gate that decides what is kept and what is forgotten from previous cell state.

$$\Gamma_f = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

5. Output gate that decides what parts of cell are output to hidden state

$$\Gamma_o = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

Using BPTT, we can show that $\forall W \in \{W_{xi}, W_{hi}, W_{xf}, W_{hf}, W_{xc}, W_{hc}, W_{xo}, W_{ho}\}$ the derivatives of the loss function are written as

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\frac{\partial c_t}{\partial c_{t-1}} \cdots \frac{\partial c_1}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\left(\prod_{j=i+1}^{t} \frac{\partial c_j}{\partial c_{j-1}}\right)\frac{\partial c_1}{\partial W}$$

To solve the vanishing or exploding gradient problem, we need $\| \frac{\partial c_j}{\partial c_{j-1}} \|_2 \approx 1$. So, let us then expand the product expression:

$$\frac{\partial c_j}{\partial c_{j-1}} = \frac{\partial}{\partial c_{j-1}}(\Gamma_i \odot \Gamma_c + \Gamma_f \odot c_{j-1}) = \frac{\partial}{\partial c_{j-1}}(\Gamma_i \odot \Gamma_c) + \frac{\partial}{\partial c_{j-1}}(\Gamma_f \odot c_{j-1}) =$$
$$= \frac{\partial \Gamma_i}{\partial c_{j-1}}\Gamma_c + \frac{\partial \Gamma_c}{\partial c_{j-1}}\Gamma_i + \frac{\partial \Gamma_f}{\partial c_{j-1}}c_{j-1} + \Gamma_f$$

We can observe that LSTM's additive property enables us to balance the gradient values during BPTT:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\left(\prod_{j=i+1}^{t} \left[\frac{\partial \Gamma_i}{\partial c_{j-1}}\Gamma_c + \frac{\partial \Gamma_c}{\partial c_{j-1}}\Gamma_i + \frac{\partial \Gamma_f}{\partial c_{j-1}}c_{j-1} + \Gamma_f\right]\right)\frac{\partial c_1}{\partial W}$$

2.3.3. Gated Recurrent Units

Cho et al.[13] proposed in 2014 an alternative to the LSTM. It has only the hidden state $h_t = (1 - \Gamma_u) \odot \Gamma_n + \Gamma_u \odot h_{t-1}$ with three gates which offers comparable performance to that of LSTM and is significantly faster to compute:

1. Reset gate:

$$\Gamma_r = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$

2. Update gate:

$$\Gamma_u = \sigma(W_{xu}x_t + W_{hu}h_{t-1} + b_u)$$

3. New gate:

$$\Gamma_n = \tanh(W_{xn}x_t + b_{xn} +_r (W_{hn}h_{t-1} + b_{hn}))$$

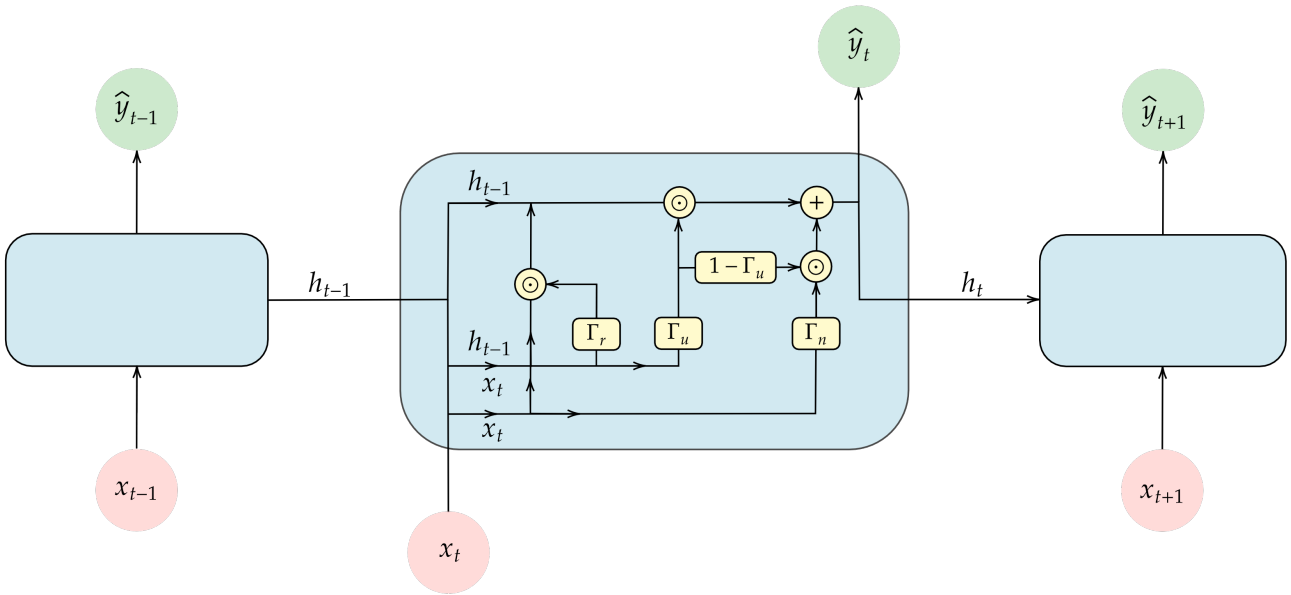

Figure 2.5 — Gated recurrent unit.

2.4. Metaheuristic Optimization

Optimal hyperparameters, Θ, of a RNN model that are set before training the neural network, such as a number of neurons and their size, are difficult to determine. Hence, we use a metaheuristic optimization technique to find a good approximation of the global minimum, in contrast to deterministic optimization methods that almost virtually find a local rather than a global optimum solution.

2.4.1. Genetic Algorithm

Genetic algorithm (GA) is a population-based metaheuristic optimization algorithm proposed by Holland [16]. The GA imitates the process of natural evolution in solving an optimization problem by relying on biologically inspired operators such as natural selection, mutation, and crossover.

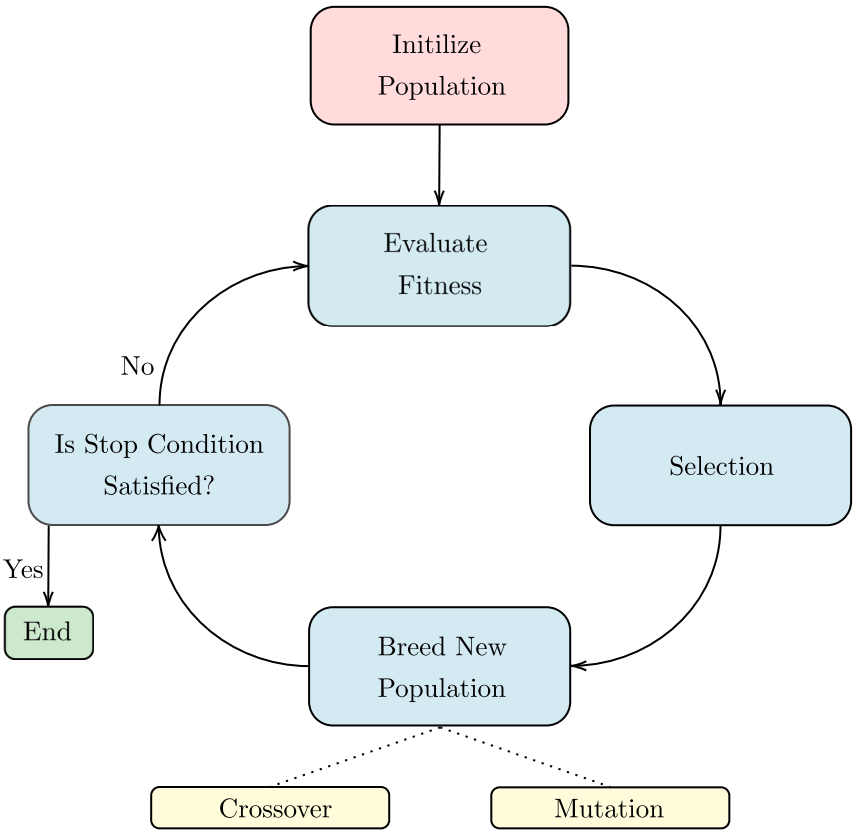

Figure 2.6 — Genetic algorithm.

Let our population Θ has ℓ individuals with chromosomes $\{\theta_i\}_{i=1}^{\ell}$, as represented in figure 2.7, where a chromosome $\theta_i \in \mathbb{R}^p$ is a set of p hyperparameters, and a hyperparameter is called a gene. Then, the population at the time step k is written as follows:

$$\Theta_k = \left\{ \theta_i^{(k)} \mid \left(\theta_{i1}^{(k)}, \ldots, \theta_{ip}^{(k)] } \right) \in \mathbb{R}^p \right\}_{i=1}^{\ell}$$

Now let us look at the algorithm itself:

1. We randomly generate the first population of individuals:

$$\Theta_0 = \left\{ \left(\theta_1^{(0)}, \ldots, \theta_\ell^{(0)} \right) \in \mathbb{R}^{\ell \times d} \mid \theta_{ij}^{(0)} \sim U(\alpha, \beta) \right\}$$

2. Then assess the fitness of each individual. For this, we need to calculate the values of the objective function for each chromosome

$$\mathcal{L}_k = \left\{ Q\left(\theta_i^{(k)} \right) \right\}_{i=1}^{\ell}$$

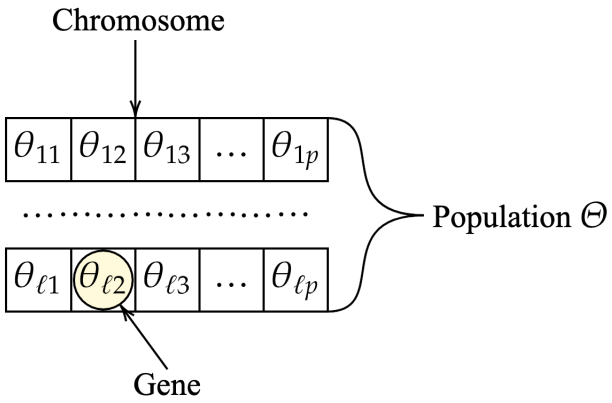

Figure 2.7.

and apply a transformation function

$$\sigma(z) = \frac{z - \mathcal{L}_k^{worst}}{\mathcal{L}_k^{best} - \mathcal{L}_k^{worst}}$$

such that its components lie between zero and one. For instance, if an individual z has near-one value, it means that it has high fitness relative to other. Hence, we get the values of the fitness function of individuals $\mathcal{F}_k = \left\{\sigma\left(\mathcal{L}_k\left(\theta_i^{(k)}\right)\right)\right\}_{i=1}^{\ell}$.

3. Select the most fitted m individuals for further breeding. For this, we assume that an individual ξ is better fitted than an individual η if $\mathcal{F}_k(\xi) = \sigma(Q(\xi)) > \sigma(Q(\eta)) = \mathcal{F}_k(\eta)$. Then, we rank individuals in the current population according to their fitness values in descending order, that is from the most fitted to the less:

$$\mathcal{F}_k^{[1]} \geqslant \ldots \geqslant \mathcal{F}_k^{[m]} \geqslant \ldots \geqslant \mathcal{F}_k^{[\ell]}$$

4. Crossover the fittest m individuals with everyone else. This is how we spread the "good" genes throughout the population. By crossover we mean the operation of creating a new chromosome, wherein a part of the genes will be from the high fitted individual $\xi^{(k)} \in \left\{\theta_i^{(k)}\right\}_{i=1}^{m}$ with the probability of

$$\mathbb{P}\left(\left\{\theta_i^{new} = \xi^{(k)}\right\}\right) = \frac{\sigma\left(Q\left(\xi^{(k)}\right)\right)}{\sigma\left(Q\left(\xi^{(k)}\right)\right) + \sigma\left(Q\left(\eta^{(k)}\right)\right)}$$

and of the genes will be from the less fitted individual $\eta^{(k)} \in \left\{\theta_i^{(k)}\right\}_{i=m+1}^{n}$ with the probability of

$$\mathbb{P}\left(\left\{\theta_i^{new} = \eta^{(k)}\right\}\right) = \frac{\sigma\left(Q\left(\eta^{(k)}\right)\right)}{\sigma\left(Q\left(\xi^{(k)}\right)\right) + \sigma\left(Q\left(\eta^{(k)}\right)\right)}$$

. Thus, we get

$$\theta_{ij}^{new} = \begin{cases} \xi_j^{(k)}, & \text{with } \mathbb{P}\left(\left\{\theta_i^{new} = \xi^{(k)}\right\}\right) \\ \eta_j^{(k)}, & \text{with } \mathbb{P}\left(\left\{\theta_i^{new} = \eta^{(k)}\right\}\right) \end{cases}, i \in (m{+}1, \ldots, n), j \in (1, \ldots, p)$$

5. All individuals except the best in the current population, $\left\{\theta_i^{(k)}\right\}_{i=2}^{\ell}$, are mutated. The mutation changes an arbitrary number of genes in an individual's chromosome for other but rather close to the original value, so afterwards we get

$$\Theta_{k+1} = \left\{\theta_1^{(k)}\right\} \cup \left\{\theta_i^{(k)} \mid \theta_{ij}^{(k)} = \theta_{ij}^{(k)} + \varepsilon_j\right\}_{i=2}^{\ell}$$

6. Go to the step 2 until the stop criterion is met. For instance, it can be a reach of the maximum possible number of population evolutions.

3. Implementation

In this paper, the Standard and Poor's 500, or simply the S&P 500, index is considered to compare the ARIMA as a benchmark model with metaheuristically optimized RNNs. The S&P500 index is a free-float weighted index of 500 of the largest companies listed on the US stock exchanges. The free float means that to calculate a firm's weight in the index, the total value of the firm's shares that are available to the investing public are examined.

Figure 3.1 — The S&P 500 index.

The research data for the index forecasting comes from 1 January 2000 to 1 January 2021 and is obtained from Yahoo Finance. Thus, an entire dataset has 5284 samples, or trading dates, in the form of daily closing prices. For the ARIMA model, we purely divide the dataset into training (90%) and test (10%) sets, while for globally optimized RNNs, we also separate the validation set (25%) from the training set in order to correctly choose the hyperparameters Θ_{RNN}.

Several objective functions to measure forecast accuracy are used in the paper, namely

$$\text{RMSE}(\hat{y}, X) = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(y_t - \hat{y}(x_t; w))^2}$$

$$\text{MAE}(\hat{y}, X) = \frac{1}{T}\sum_{t=1}^{T}|y_t - \hat{y}(x_t; w)|$$

$$\text{SMAPE}(\hat{y}, X) = \frac{1}{T}\sum_{t=1}^{T}\frac{|y_t - \hat{y}(x_t; w)|}{(|y_t| + |\hat{y}(x_t; w)|)/2}$$

and also one performance metric is used

$$R^2\left(\hat{y}, X\right) = 1 - \frac{\sum\limits_{t=1}^{T}(y_t - \hat{y}(x_t; w))^2}{\sum\limits_{t=1}^{T}(y_t - \overline{y})^2}, \overline{y} = \frac{1}{T}\sum_{t=1}^{T} y_t$$

3.1. Linear Time Series Forecasting

For this section, the Box-Jenkins methodology [18] is used to construct the ARIMA model. At the first stage, we use a Box-Cox transformation to stabilize the variance of the series with $\lambda = 0$:

$$\tilde{y}_t = \begin{cases} \ln y_t & \lambda = 0 \\ \dfrac{y_t^{\lambda} - 1}{\lambda}, & \lambda \neq 0 \end{cases}$$

where y_t is a closing price of S&P500 index at time step t.

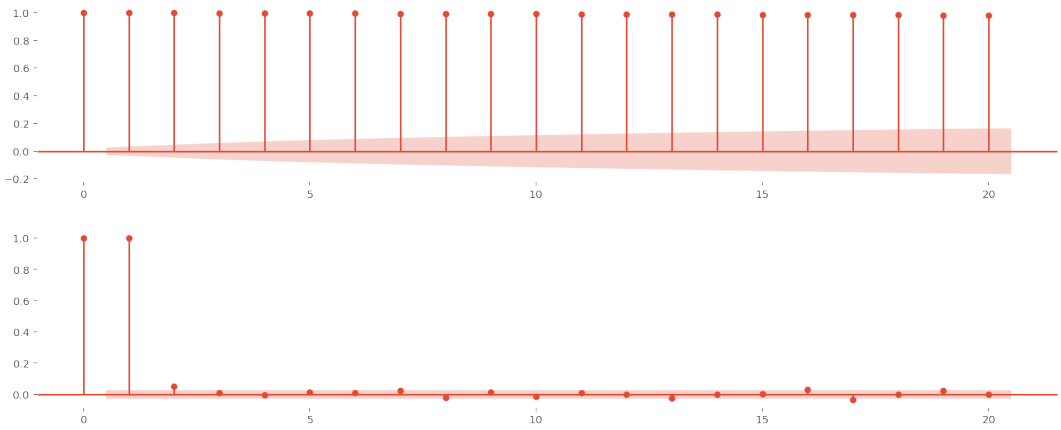

Figure 3.2 — Autocorrelation (top) and partial autocorrelation (bottom) of the S&P500 index.

Then we identify whether our time series is stationary or not with the autocorrelation (ACF) and partial autocorrelation (PACF) functions. The former measures the linear relationship between the value of the index at the time step t with that of lagged value at the time step $t - k$:

$$\hat{\rho}_k = \widehat{\mathbb{C}orr}[y_t, y_{t-k}] = \frac{\sum\limits_{t=k+1}^{T}(y_t - \overline{y})(y_{t-k} - \overline{y})}{\sqrt{\sum\limits_{t=k+1}^{T}(y_t - \overline{y})^2 \sum\limits_{t=k+1}^{T}(y_{t-k} - \overline{y})^2}}, \overline{y} = \frac{1}{T}\sum_{t=1}^{T} y_t, 0 \leqslant k \leqslant T-1$$

Whereas the latter is a «pure» correlation, noted β_k, that can be obtained from the AR(k) process:

$$y_t = \beta_0 + \beta_1 y_{t-1} + \cdots + \beta_k y_{t-k} + \varepsilon_t$$

Note that, if we assume the series $\{y_t\}_t^T$ is independently and identically distributed, it can be shown that $\hat{\rho}_k\sqrt{T} \xrightarrow{\mathcal{D}} \mathcal{N}(0,1)$. Therefore, the 95-% confidence interval for the insignificant autocorrelation is represented as $\left(-\frac{1.96}{\sqrt{T}}, \frac{1.96}{\sqrt{T}}\right)$.

As we can observe in figure 3.2, the time series is clearly not stationary. Hence, we conclude that differencing is required and use a first-order difference, $\Delta y_t = y_t - y_{t-1}$, as shown in figure 3.3. After using differencing, it can easily be noted from figure 3.4 that the time series $\{\Delta y_t\}_{t=1}^T$ appears to be stationary both in its mean and variance.

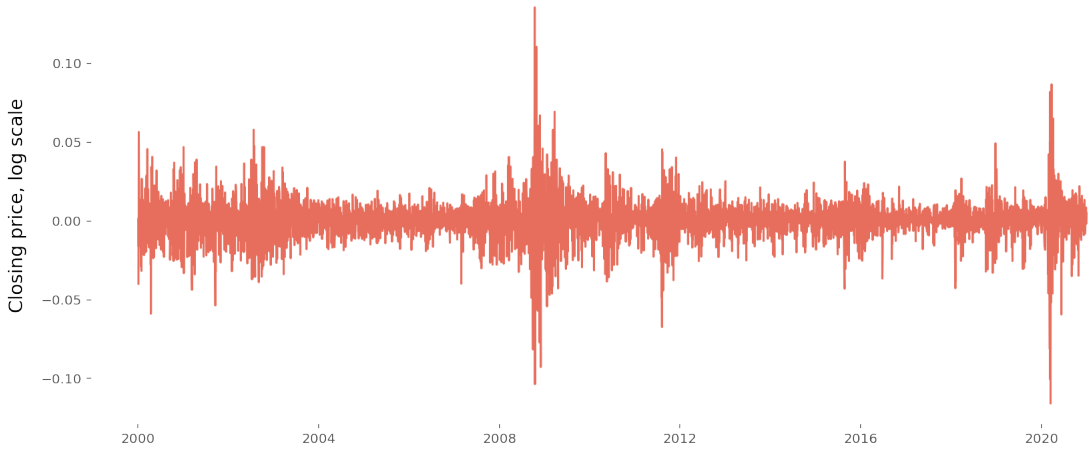

Figure 3.3 — The first-order differenced S&P 500 index.

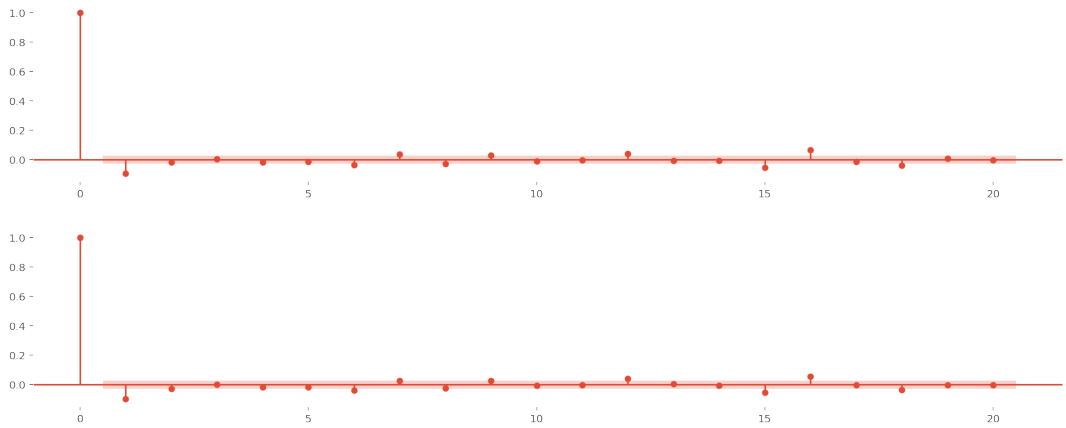

Figure 3.4 — Autocorrelation (top) and partial autocorrelation (bottom) of the first-order S&P500 index.

Nevertheless, to determine more objectively whether additional differencing is required, an augmented Dickey-Fuller (ADF) test is used, which is a part of unit root tests. Our null hypothesis is that the time series is non-stationary, while the alternative hypothesis that it is stationary. For our data, the ADF statistic is -13.649 and its p-value is 1.6e-25. Thus, we fail to accept the null hypothesis and conclude that the series is stationary.

This allows us to choose a number of p AR and q MA lags for the ARIMA model. For that we examine the stationary first-order differenced time series shown in figure 3.4 and propose several models. The best model is then chosen by the Akaike (AIC), corrected Akaike (AICc), and Bayes (BIC) information criteria:

$$\begin{aligned}
\text{AIC} &= 2n - 2\ln\mathcal{L} \\
\text{AICc} &= AIC + \frac{2n(1+n)}{T-s-n-1} \\
\text{BIC} &= n\ln(T-s) - 2\ln\mathcal{L}, \;\; s = \max(p_{max}, q_{max})
\end{aligned}$$

where n is a number of model parameters and $\mathcal{L}$ is likelihood.

However, in our case, we just can use the automatic ARIMA implemented in Python, which itself selects the correct number of lags by minimizing the above-noted information criteria. Its prediction results are shown in figure 3.5.

Figure 3.5 — Automatic ARIMA for the S&P500 index.

In the end, to assure that our model is adequate, we investigate if a series of the forecast errors can be considered as a white noise. To check that, the Ljung-Box test is used, where the null hypothesis assumes there is no remaining residual autocorrelation at lags 1 to k, while the alternative hypothesis assumes that at least

one of the autocorelations is non-zero:

$$\begin{cases} H_0: & \rho_1 = \ldots = \rho_k = 0 \\ H_a: & \exists \rho_i \neq 0,\ i \in \{1, \ldots, k\} \end{cases}$$

The Ljung-Box statistic has the form of $Q_{obs} = T(T+2)\sum_{s=1}^{k} \frac{\hat{\rho}_s^2}{T-s} \overset{H_0}{\sim} \chi^2(k)$, where k is a number of lags being tested. For our data, considering 10 lags the Ljung-Box statistic is equal to 4893.6 with p-value of 0. Thus, the null hypothesis is rejected and we assume that there is a significant residual autocorrelation. Hence, our model is not fully adequate, and a different model should be considered.

3.2. Non-Linear Time Series Forecasting

Since the ARIMA model has no significant performance, the GA-optimized recurrent network, or simply GA-RNN, with 14 look-back periods is proposed, as depicted in figure 3.6.

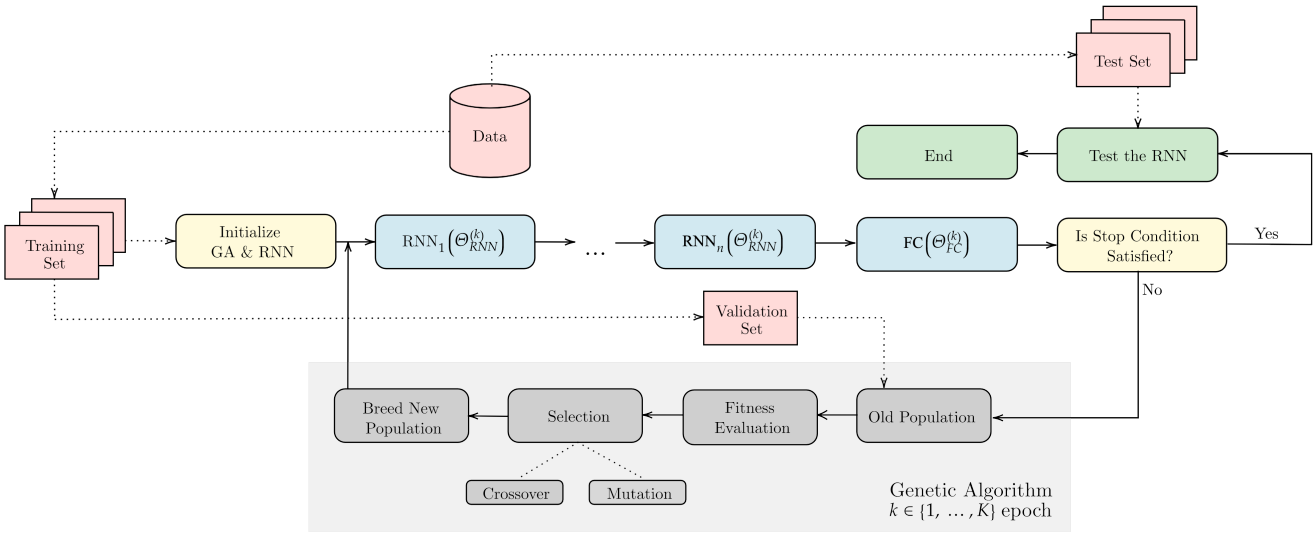

Figure 3.6 — Recurrent neural network architecture with the use of genetic algorithm as a metaheuristic optimizer.

Given the GA-RNN architecture, we can add additional features into our dataset, namely trading volume for the S&P500 index and its technical indicators, such as 14-day simple and weighted moving averages, 200-day exponential moving average, relative strength index, and bollinger bands. So, in this subsection, we consider four models: GA-LSTM and GA-GRU that take only a series of closing

prices as an input, plus GA-LSTM-TI and GA-GRU-TI that take a multivariate series with technical indicators.

Figure 3.7 — Buy and sell signals of relative strength index (top) and bollinger bands (bottom).

In addition to that, we introduce dummy variables that are considered to be buy and sell signals. They are supposed to help the GA-RNN-TI models predict the right closing price movement. Let $i \in \{\text{SMA}, \text{WMA}, \text{EMA}\}$ and $j \in \{\text{RSI}, \text{BB}\}$, then dummy variables can be written as

$$D_{it}^{\text{below}} = \begin{cases} 1, & \text{closing price is below } n\text{-day indicator } i \text{ at time } t \\ 0, & \text{otherwise} \end{cases}$$

$$D_{it}^{\text{above}} = \begin{cases} 1, & \text{closing price is above } n\text{-day indicator } i \text{ at time } t \\ 0, & \text{otherwise} \end{cases}$$

$$D_{jt}^{\text{oversold}} = \begin{cases} 1, & n\text{-day indicator } j \text{ at time } t \text{ is lower than } 30 \\ 0, & \text{otherwise} \end{cases}$$

$$D_{jt}^{\text{overbought}} = \begin{cases} 1, & n\text{-day indicator } j \text{ at time } t \text{ is greater than } 70 \\ 0, & \text{otherwise} \end{cases}$$

The indicator variables D_{jt}^{oversold} and $D_{jt}^{\text{overbought}}$ are depicted in figure 3.7.

After creating dummy variables, the multivariate time-series are min-max normalized, such that the preprocessed data lies between zero and one:

$$\tilde{x}_{tj} = \frac{x_{tj} - \min_t x_{tj}}{\max_t x_{tj} - \min_t x_{tj}}$$

Now, we attempt to choose hyperparameters of our models, that is the correct number of hidden layers and training epochs, the hidden size of neurons, and the learning rate. For this reason, the genetic algorithm is used as a metaheuristic optimizer. Hence, we create a population with 70 individuals—in our case, Adam optimized neural networks—with the mutation rate of 40% and 10 meta epochs considered. Moreover, we select the 10 most fitted individuals to crossover.

Given that a number of hidden layers can range from 1 to 15, training epochs from 60 to 300, a hidden size from 8 to 512, and a learning rate from 0.001 to 1, the result of metaheuristic and deterministic optimization, presented in figures 3.8, 3.9, and 3.10, is as follows:

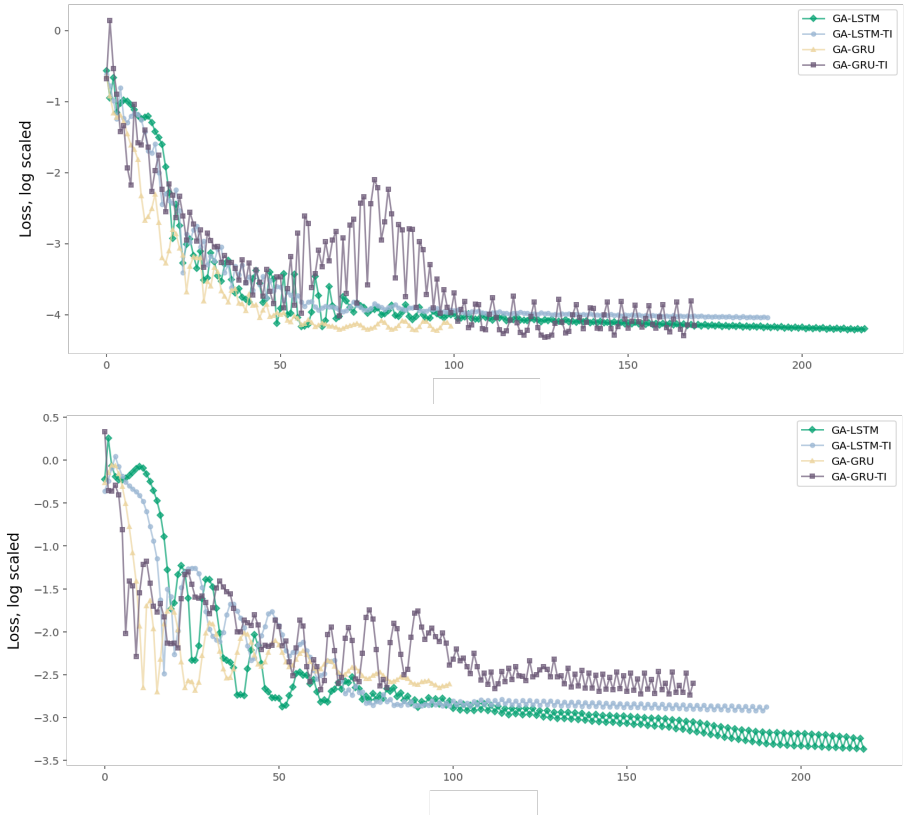

Figure 3.8 — Adam optimization process of the best individual in population for RMSE by epoch using train set (top) and test set (bottom).

1. GA-LSTM: the number of hidden layers of 2, 219 epochs, the hidden size of 64, and learning rate of 0.01.
2. GA-LSTM-TI: the number of hidden layers of 3, 191 epochs, the hidden size of 256, and learning rate of 0.001.

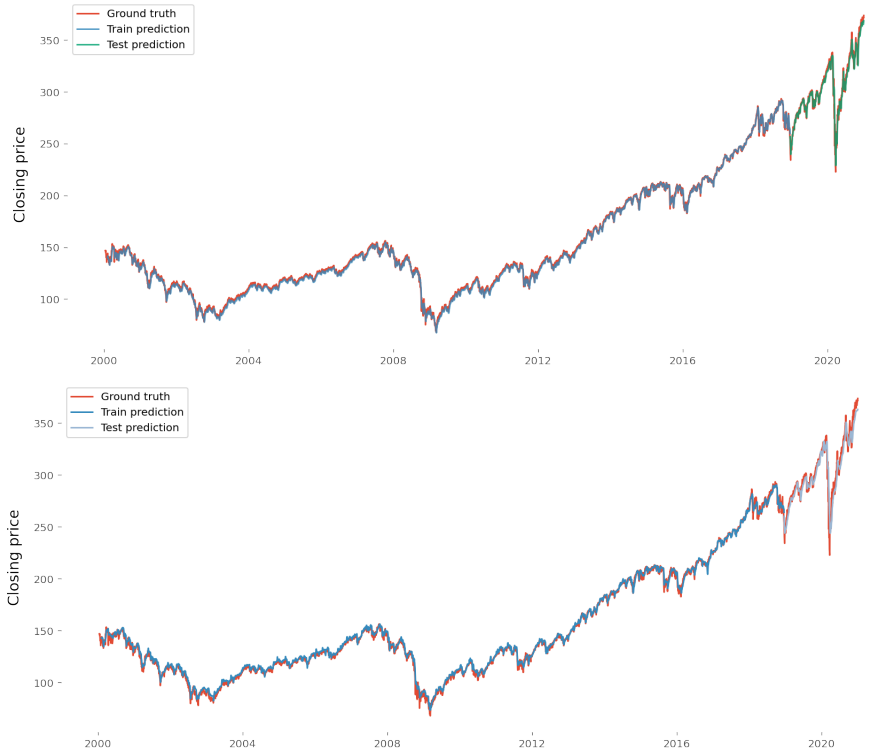

Figure 3.9 — GA-LSTM (top) and GA-LSTM-TI (bottom).

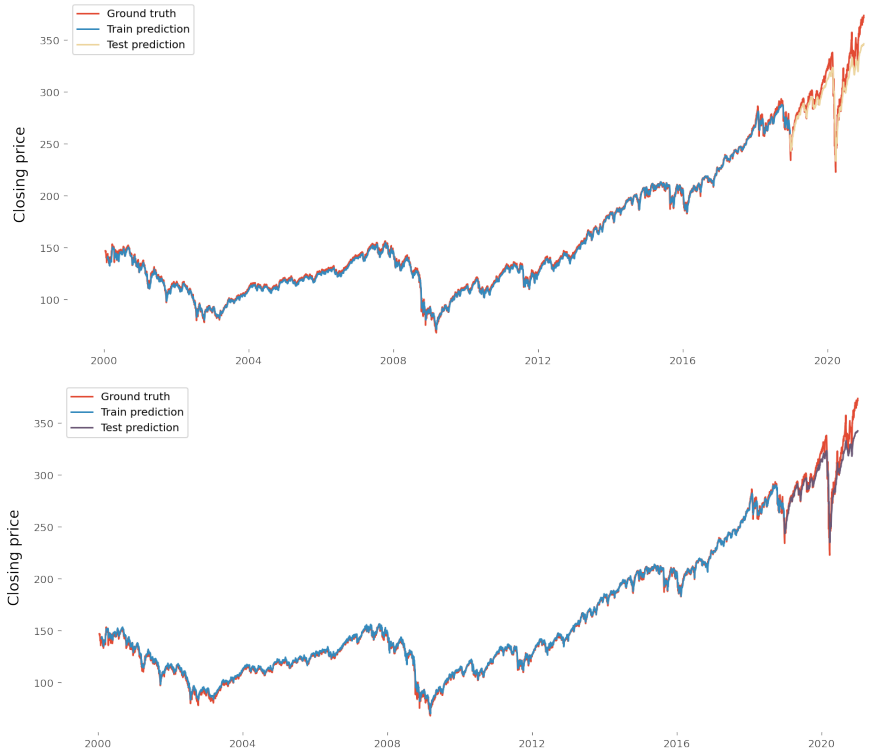

Figure 3.10 — GA-GRU (top) and GA-GRU-TI (bottom).

3. GA-GRU: the number of hidden layers of 2, 101 epochs, the hidden size of 16, and learning rate of 0.01.
4. GA-GRU-TI: the number of hidden layers of 3, 166 epochs, the hidden size of 64, and learning rate of 0.011.

4. Discussion and Conclusions

As can be observed from table 4.1, the best ARIMA model has negative R^2, which indicates that it is a wrong model and not suited for our task. On the contrary, all GA-optimized RNNs show outstanding performance on the test set.

The RMSE, MAE, and SMAPE of the GA-LSTM model are smaller than those of the other models. Its R^2 is the highest and is equal to 0.97, which means excellent prediction performance. Compared to the ARIMA model as a benchmark, the RMSE, MAE, and SMAPE of the GA-LSTM model are reduced by 715.86%, 836.74%, and 812.80%, respectively. Even in the worst-case scenario, that is considering the GA-GRU model, the value of objective functions are reduced by 333.89%, 345.91%, and 349.14%, respectively. It shows us that the GA-RNN models can effectively improve prediction performance.

Model	RMSE	MAE	SMAPE	R^2
AutoARIMA	853.31	16107.30	10.16	-0.52
GA-LSTM	119.20	1925	1.25	0.97
GA-LSTM-TI	195.20	3276.67	2.13	0.92
GA-GRU	255.57	4656.46	2.91	0.86
GA-GRU-TI	256.99	4356.50	2.71	0.86

Table 4.1 — Prediction error and performance.

However, it can also be seen from the table 4.1 that the introduction of technical indicators as additional features did not pose a positive effect. This is most likely the reason that we used a small amount of them. After all, the strength of technical indicators is when they all show the same or near the same signal for the data. Hence, the impact of technical indicators on the model's forecast should be studied in more detail by using an ampler amount of such indicators, which is out of the scope of this paper.

Appendices

Packages

```
In:   import pandas as pd
      import numpy as np
      from math import sqrt
      import random

      import statsmodels.api as sm
      from statsmodels.tsa.stattools import adfuller
      from statsmodels.tsa.stattools import kpss
      from sktime.forecasting.arima import ARIMA, AutoARIMA
      from sktime.forecasting.base import ForecastingHorizon

      import torch
      from torch import nn
      from torch import optim

      from dataclasses import dataclass

      import gc
      import os

      device = torch.device('cuda') if torch.cuda.is_available()
               else torch.device('cpu')

      def seed_everything(seed: int = 77):
          random.seed(seed)
          os.environ['PYTHONHASHSEED'] = str(seed)
          np.random.seed(seed)

          torch.manual_seed(seed)
          torch.cuda.manual_seed(seed)
          torch.cuda.manual_seed_all(seed)

          torch.backends.cudnn.enabled = False
          torch.backends.cudnn.deterministic = True

      os.environ['CUBLAS_WORKSPACE_CONFIG'] = ':4096:2'
```

Technical Indicators

In:
```
def simple_moving_average(df: pd.DataFrame, n: int = 10,
                          prices: str = 'Close') -> pd.Series:
    return df[prices].rolling(window=n, min_periods=n).mean()


def weighted_moving_average(df: pd.DataFrame, n: int = 10,
                            prices: str = 'Close') -> pd.Series:
    return df[prices].rolling(window=n,
                              min_periods=n).apply(
                              lambda x:x[::-1].cumsum().sum()*2/n/(n+1))


def exponential_moving_average(df: pd.DataFrame, n: int = 10,
                               prices: str = 'Close') -> pd.Series:
    return df[prices].ewm(span=n, min_periods=n).mean()


def relative_strength_index(df: pd.DataFrame, n: int,
                            prices: str = 'Close') -> pd.Series:
    deltas = df[prices].diff()
    ups = deltas.clip(lower=0)
    downs = (-deltas).clip(lower=0)

    numerator = ups.ewm(com=n-1, min_periods=n).mean()
    denominator = downs.ewm(com=n-1, min_periods=n).mean()
    rs =  numerator/ denominator
    rsi = 100 - 100 / (1 + rs)
    return rsi


def bollinger_bands(df: pd.DataFrame, n: int = 20,
                    m: float = 2.0) -> Tuple[pd.Series]:

  typical_price = (df['High'] + df['Low'] + df['Close']) / 3

  sma = typical_price.rolling(window=n, min_periods=n).mean()
  sigma = typical_price.rolling(window=n, min_periods=n).std()

  upper_bollinger_band = sma + m * sigma
  lower_bollinger_band = sma - m * sigma

  return lower_bollinger_band, upper_bollinger_band
```

Neural Network Training

In:
```
def train_one_epoch(model, X, y_true, criterion, optimizer):
    model.train()

    y_pred = model(X.to(device))
    loss = criterion(y_pred.unsqueeze(1),
                     y_true.unsqueeze(1).to(device))
    train_loss = loss.item()

    nn.utils.clip_grad_norm_(model.parameters(), 5)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    return train_loss


def predict(model, X, y_true, criterion):
    model.eval()

    y_pred = model(X.to(device))
    loss = criterion(y_pred.unsqueeze(1),
                     y_true.unsqueeze(1).to(device))
    test_loss = loss.item()

    return test_loss


def train(model, criterion, optimizer, X_train, y_train,
          X_test=None, y_test=None, n_epochs=10, verbose=True,
          return_loss_history=True, compute_test_loss=True):
    model.to(device)

    history_train_loss = []
    history_test_loss = []

    for epoch in range(n_epochs):
        history_train_loss.append(train_one_epoch(model,
                                                  X_train,
                                                  y_train,
                                                  criterion,
                                                  optimizer))
```

```
            if compute_test_loss:
                history_test_loss.append(predict(model,
                                                 X_test,
                                                 y_test,
                                                 criterion))
            if verbose:
                clear_output(wait=True)
                print(f'Epoch: {epoch + 1}')

                plot_metric(f'{criterion.__class__.__name__}',
                            history_train_loss,
                            history_test_loss)

                print(f'Train loss: {history_train_loss[-1]:.4}')

                if compute_test_loss:
                    print(f'Test loss: {history_test_loss[-1]:.4}')

        if return_loss_history:
            return history_train_loss, history_test_loss
```

Long Short-Term Memory

```
In:   class LSTM(nn.Module):
        def __init__(self, input_size: int = 1, hidden_size: int = 32,
                     num_layers: int = 2, dropout: float = 0):
            super(LSTM, self).__init__()
            self.input_size = input_size
            self.hidden_size = hidden_size
            self.num_layers = num_layers

            self.lstm = nn.LSTM(input_size=input_size,
                                hidden_size=hidden_size,
                                num_layers=num_layers,
                                dropout=dropout,
                                batch_first=True)

            self.fc = nn.Linear(hidden_size, 1)


        def forward(self, x):
            h_0 = torch.zeros(self.num_layers, x.size(0),
                              self.hidden_size).to(device).requires_grad_()
```

```
        c_0 = torch.zeros(self.num_layers, x.size(0),
                          self.hidden_size).to(device).requires_grad_()

        out, (h_n, c_n) = self.lstm(x, (h_0.detach(), c_0.detach()))

        out = self.fc(out[:, -1, :])

        return out
```

Gated Recurrent Units

```
In:  class GRU(nn.Module):
    def __init__(self, input_size: int = 1, hidden_size: int = 32,
                 num_layers: int = 2, dropout: float = 0):
        super(GRU, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.gru = nn.GRU(input_size=input_size,
                          hidden_size=hidden_size,
                          num_layers=num_layers,
                          dropout=dropout,
                          batch_first=True)

        self.fc = nn.Linear(hidden_size, 1)


    def forward(self, x):
        h_0 = torch.zeros(self.num_layers, x.size(0),
                          self.hidden_size).to(device).requires_grad_()
        out, (h_n) = self.gru(x, (h_0.detach()))

        out = self.fc(out[:, -1, :])

        return out
```

Genetic Algorithm

```
In:   @dataclass
      class GeneticAlgorithmConfig:
          ell: int = 20
          k: int = 5

          mutation_rate: float = 0.1
          num_epochs: int = 10

      class Individual:
          def __init__(self):
              self.name = '#'+''.join(map(str,
                                          np.random.randint(0, 9,
                                                            size=7).tolist()))
              self.num_epochs_base = np.random.choice(np.arange(10, 120))
              self.hidden_size = np.random.choice([2**p for p in range(2, 10)])
              self.num_layers = np.random.choice(np.arange(2, 15))
              self.learning_rate = round(np.random.random(), 2)

              self.loss = np.inf
              self.fitness = None


      @dataclass
      class Population:
          def __init__(self, config: GeneticAlgorithmConfig):
              self.individuals = [Individual() for _ in range(config.ell)]
              self.top_k_individuals = None
              self.best_indivdual = None


      class GeneticAlgorithm:
          def __init__(self, optimized_block, criterion,
                       population: Population, config: GeneticAlgorithmConfig,
                       device, verbose=True, seed: int = 77):
              self.optimized_block = optimized_block
              self.criterion = criterion
              self.population = population
              self.config = config
              self.device = device
              self.verbose = verbose
              self.seed = seed
```

```
        self.loss_hist = []


    def fit(self, X_val, y_val):
        pop = self.population
        config = self.config

        for epoch in range(config.num_epochs):
            self.evaluate(X_val, y_val, pop)
            self.select(pop)
            self.loss_hist.append(pop.best_indivdual.loss)

            offsprings = []
            for weak_ind in pop.individuals[config.k:]:
                strong_ind = np.random.choice(pop.top_k_individuals)
                offsprings.append(self.crossover(strong_ind, weak_ind))

            new_pop = pop.top_k_individuals + offsprings

            mutated_pop = []
            for individual in new_pop[1:]:
                mutated_pop.append(self.mutate(individual))

            pop.individuals = [pop.best_indivdual] + mutated_pop

            if self.verbose:
                clear_output(wait=True)
                print(f"Epoch: {epoch + 1}")

                plot_metric(self.criterion.__class__.__name__,
                            val_metric=self.val_loss_history)

                print(f'{pop.best_indivdual}')


    def evaluate(self, X_val, y_val, population: Population):
        losses = []

        for indiv in population.individuals:
            gc.collect()
            torch.cuda.empty_cache()

            if self.optimized_block == 'LSTM':
                seed_everything(self.seed)
```

```
                model = LSTM(input_size=X_val.shape[2],
                             hidden_size=int(indiv.hidden_size),
                             num_layers=indiv.num_layers).to(self.device)

            elif self.optimized_block == 'GRU':
                seed_everything(self.seed)
                model = GRU(input_size=X_val.shape[2],
                            hidden_size=int(indiv.hidden_size),
                            num_layers=indiv.num_layers).to(self.device)

            else:
                raise ValueError('Only LSTM and GRU blocks are
                                  available for optimization.')

            optimizer = optim.Adam(model.parameters(),
                                   lr=indiv.learning_rate)

            seed_everything(self.seed)
            train(model, self.criterion, optimizer, X_val, y_val,
                  individual.num_epochs_base,
                  verbose=False, return_loss_history=False,
                  compute_test_loss=False)

            individual.loss = predict(model, X_val, y_val,
                                      self.criterion)

            losses.append(indiv.loss)

            del model

        for indiv in population.individuals:
            indiv.fitness = self.normalize(indiv.loss,
                                           min(losses),
                                           max(losses))


    def normalize(self, z, loss_best, loss_worst) -> float:
        return (z - loss_worst) / (loss_best - loss_worst)


    def select(self, population: Population):
        ranked_population = sorted(population.individuals,
                                   key=lambda idiv: idiv.fitness,
                                   reverse=True)
```

```
        population.best_indivdual = ranked_population[0]

        population.top_k_individuals = ranked_population[:self.config.k]


    def crossover(self, s_parent: Individual,
                  w_parent: Individual) -> Individual:
        offspring = Individual()

        prob = s_parent.fitness / (s_parent.fitness + w_parent.fitness)

        if np.random.random() > prob:
            offspring.hidden_size = w_parent.hidden_size
        else:
            offspring.hidden_size = s_parent.hidden_size

        if np.random.random() > prob:
            offspring.num_layers = w_parent.num_layers
        else:
            offspring.num_layers = s_parent.num_layers

        if np.random.random() > prob:
            offspring.learning_rate = w_parent.learning_rate
        else:
            offspring.learning_rate = s_parent.learning_rate

        if np.random.random() > prob:
            offspring.num_epochs_base = w_parent.num_epochs_base
        else:
            offspring.num_epochs_base = s_parent.num_epochs_base

        return offspring


    def mutate(self, indiv: Individual) -> Individual:
        if np.random.random() < self.config.mutation_rate:
            rand = np.random.randint(-1, 2))
            indiv.hidden_size = 2 ** (np.log2(indiv.hidden_size) + rand
            a, b = 2 ** 3, 2 ** 9
            hidden_size_ = int(np.array(indiv.hidden_size).clip(a, b))
            indiv.hidden_size = hidden_size_

        if np.random.random() < self.config.mutation_rate:
            rand = np.random.randint(-2, 3)
            indiv.num_layers += rand
```

```
            a, b = 2, 14
            num_layers_ = np.array(indiv.num_layers).clip(a, b)
            indiv.num_layers = num_layers_

        if np.random.random() < self.config.mutation_rate:
            rand = round(np.random.uniform(-0.1, 0.1), 2)
            indiv.learning_rate += rand
            a, b = 0.001, 1
            learning_rate_ = np.array(indiv.learning_rate).clip(a, b)
            indiv.learning_rate = learning_rate_

        if np.random.random() < self.config.mutation_rate:
            rand = np.random.randint(-10, 10)
            indiv.num_epochs_base += rand
            a, b = 10, 120
            num_epochs_base_ = np.array(indiv.num_epochs_base).clip(a,b)
            indiv.num_epochs_base = num_epochs_base_

        return indiv
```

References

[1] Wang, P., Zhao, J., Gao, Y., Sotelo, M. A., and Li, Z. (2020). "Lane work-schedule of toll station based on queuing theory and PSO-LSTM model". In: *IEEE Access*, 8, pp. 84434-84443.

[2] Ji, Y., Liew, A. W., and Wang, L. Y. (2021). "A novel improved particle swarm optimization with Long-Short Term Memory hybrid model for stock indices forecast". In: *IEEE Access*, 9, pp. 23660–23671.

[3] Chung, H., and Kyung-shik, S. (2018). "Genetic algorithm-optimized Long Short-Term Memory network for stock market Ppediction". In: *Sustainability*.

[4] Wold, H. (1954). "A study in the analysis of stationary time series". In: *Journal of the Royal Statistical Society* 102.2, pp. 295-298.

[5] Hyndman, R. J., Athanasopoulos, G. (2021). "Forecasting: principles and practice".

[6] Tsay, R. S. (1951). "Analysis of financial time series".

[7] Nesterov, Y. (2018). "Lectures on convex optimization".

[8] Duchi, J., Hazan, E., and Singer, Y. (2011). "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research*, 12.7, pp. 2121–2159.

[9] Tieleman, T., and Hinton, G. (2012). "Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude". In: *Coursera*, pp. 26–31.

[10] Kingma, D. P., and Ba, J. (2014). "Adam: a method for stochastic optimization". In: *ArXiv*.

[11] Pascanu, R., Mikolov, T., and Bengio, Y. (2012). "On the difficulty of training Recurrent Neural Networks". In: *ArXiv*.

[12] Hochreiter, S., and Schmidhuber, J. (1997). "Long short-term memory". In: *Neural computation*, 9.8, pp. 1735–1780.

[13] Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). "On the properties of neural machine translation: encoder-decoder approaches". In: *ArXiv*.

[14] Berner, J., Grohs, P., Kutyniok, G., and Petersen, P. (2021). "The modern mathematics of deep learning". In: *ArXiv*.

[15] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2020). "Dive into Deep Learning"

[16] Holland, J. (1984). "Genetic algorithms and adaptation". In: *Adaptive Control of Ill-Defined Systems*.

[17] Weise, T. (2009). "Global optimization algorithms: theory and application".

[18] Warren, L. (1977). "The Box-Jenkins approach to time series analysis and forecasting : principles and applications". In: *Recherche operationnelle*, 11.2, pp. 129–143.