CSEE 4840 Design Document - FASTRADE

Spring 2024

Yixuan Li (yl5468), Wenbo Liu (wl2927), Weitao Lu (wl2928), Xiaolei Zhao (xz3283)

Contents

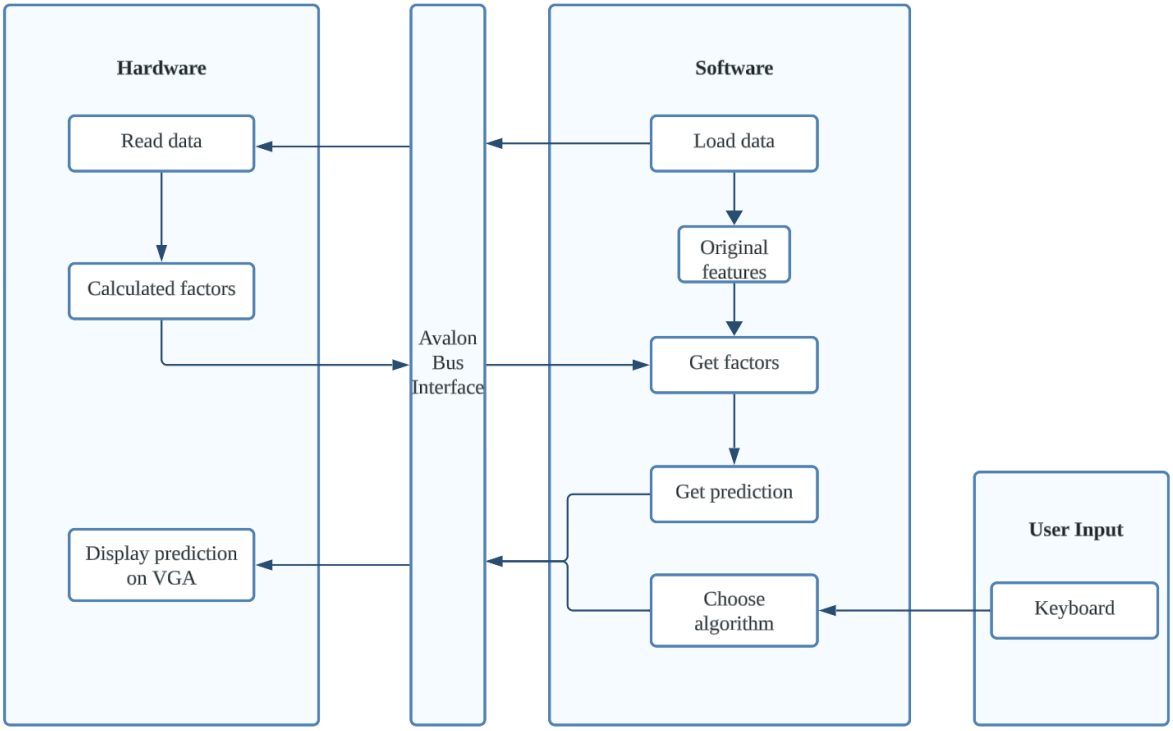
1 Introduction

The rapid fluctuations in the stock market require high-speed data processing to make informed investment decisions. Traditional computing methods can lag behind real-time requirements, leading to missed opportunities.

FASTRADE (**FPGA-Accelerated Strategy & Trading for Enhanced Decisions)** designs and implements a system that uses Field Programmable Gate Arrays (FPGA) to calculate investment factors based on daily stock prices and generate

investment strategies on these factors. The goal is to accelerate the processing of financial data to offer real-time investment insights. By leveraging FPGA for computation and software for data handling and analysis, this project aims to optimize the efficiency and accuracy of stock market investment decisions.

2 System Block Diagram



System Block Diagram for FASTRADE

The block diagram provides a structured overview of the FASTRADE system, delineating the distinct roles of hardware and software components and their interaction.

- The hardware side is responsible for reading data and processing it to distinguish between original features and calculated factors. It also finally completes the loop by displaying the prediction on a VGA output, providing a visual representation of the predictive analytics generated by the system.
- Communication between the hardware and software components is facilitated by the Avalon Bus Interface. This interface ensures that the calculated factors

are transmitted securely and efficiently to the software side for further analysis.

- On the software side, the process begins with the loading of data. It is here that the software retrieves the factors processed by the hardware. The software is then responsible for generating predictions based on these factors, demonstrating its analytical role in the system.
- User interaction with the system is managed through a separate input block, where the user provides commands via a keyboard. This allows the user to choose algorithms.

3 Algorithms

3.1 Factors:

3.1.1 Price Momentum Factor:

First, we will choose a time frame for measuring momentum. Then calculate the asset over the chosen time frame with the following formula:

$$Price\ Momentum = (Price_{end}\ /\ Price_{start}) - 1$$

For comparison purposes, we will normalize the momentum values. This can be done by dividing the momentum of each asset by the standard deviation of all momentum values.

3.1.2 Volume Factor:

First, we will choose a time frame to calculate the volume factor. For the average volume calculation, we will use the following formula:

$$Volume\ Factor = \frac{\Sigma\ Volume_i}{N}$$

For the volume ratio calculation, we use the following formular:

$$Volume\ Ratio = \frac{Volume_{current}}{Volume_{average}}$$

3.1.3 Volatility Factor:

We also need to choose a time frame before calculating. For each day in the chosen time frame, calculate the return of the security or index. The return can be calculated as the percentage change in price from the previous day using the formula:

$$Volatility\ Factor = \sqrt{\frac{\Sigma\ (Return_{current} - Return_{average})^2}{N-1}}$$

The Volatility Factor is calculated as the standard deviation of the returns over the chosen time frame. It measures the price fluctuations of a security or market index.

3.1.4 Relative Strength Index

RSI is a momentum oscillator that measures the speed and change of price movements. RSI values range from 0 to 100 and are typically used to identify overbought or oversold conditions in a traded asset. An asset is usually considered overbought when the RSI is above 70 and oversold when it's below 30.

The formula for calculating the Relative Strength Index (RSI) is:

$$RSI = 100 - (\frac{100}{1+RS})$$

where **RS** (Relative Strength) is the ratio of the average gain of the periods that closed up to the average loss of the periods that closed down. These averages are typically calculated over a 14-day period, which is the standard period used by J. Welles Wilder when he introduced the indicator, but the period can be adjusted to suit different trading strategies and time frames.

The code is like this:

```
import pandas as pd

# Assuming 'df' is a DataFrame containing your stock's price dat
df['delta'] = df['close'].diff() # Step 1: Calculate daily retu
df['gain'] = df['delta'].clip(lower=0) # Step 2: Isolate gains
```

```
df['loss'] = -df['delta'].clip(upper=0) # Step 2: Isolate losses

# Step 3: Calculate the averages of the gains and losses
avg_gain = df['gain'].rolling(window=14, min_periods=14).mean()
avg_loss = df['loss'].rolling(window=14, min_periods=14).mean()

# Step 4: Calculate RS
rs = avg_gain / avg_loss

# Step 5: Calculate RSI
df['RSI'] = 100 - (100 / (1 + rs))
```

We will rewrite the code in system verilog and run it on the FPGA.

3.1.5 Moving Average

First, decide how many periods (days, weeks, minutes, etc.) we want to include in your moving average. Then sum up the closing prices of the stock for the last N periods. Finally, divide the total sum of the closing prices by N.

The formula is as following:

$MA = \frac{\Sigma\ Price_i}{N}$

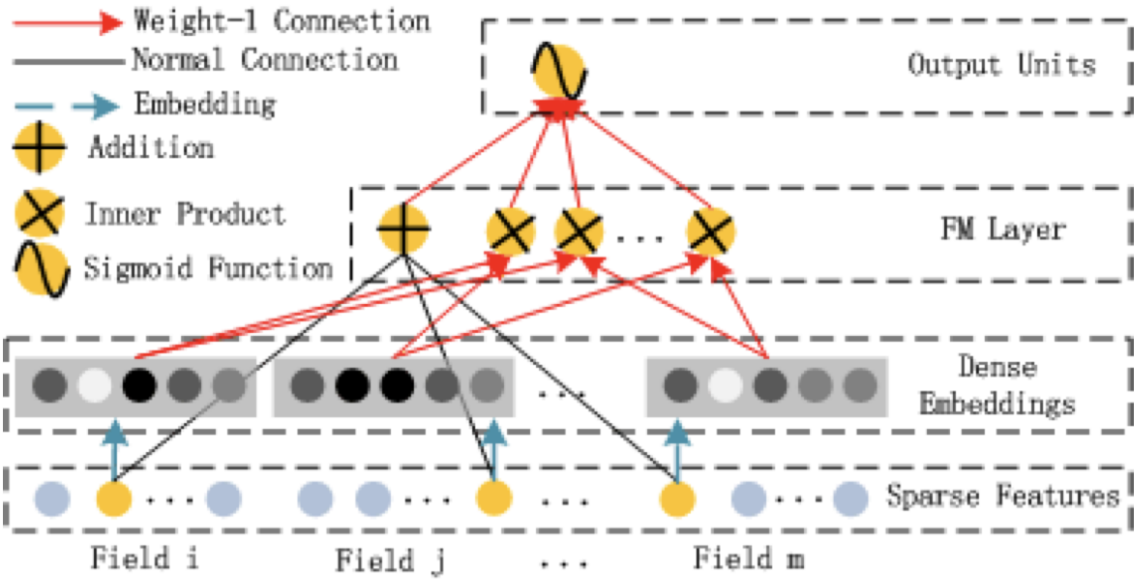
3.2 Factor Model

3.2.1 Model Overview:

Based on the calculated factors and the feature of our data, we can use a factor model to add weight to each factor and generate an output. Our factor model follows this structure:

Reference:

- [Rendle, 2010] Steffen Rendle. Factorization machines. In ICDM, 2010.
- **DeepFM: A Factorization-Machine based Neural Network for CTR Prediction**



Factor Models(FM) is a standard multifactor weighting model in factor investment and recommend systems. Besides a simple linear (order-1) interactions among features, FM models pairwise (order-2) feature inter- actions as inner product of respective feature latent vectors.

It follows 3 steps:

Given the space features, it will embed the feature into same format, and then use a FM layer to give the embedded vectors different weights. Finally it will generate an output that can be used for stock ranking or sell/buy strategy.

3.2.2 Factor Embedding:

Reference:

- [Zhang *et al.*, 2016] Weinan Zhang, Tianming Du, and Jun Wang. Deep learning over multi-field categorical data - - A case study on user response prediction. In *ECIR*, 2016.

Different Factors have different distribution, semantic meanings, some are discrete and some are continuous. So it's necessary to embed these factors into

the same dense dimension before the calculation in FM models.

The typical embedding methods are:

0.One-Hot coding:

Good for discrete data, but sparse.

1.Linear Transformation:
Embedding through a linear transformation. Given a predefined weight matrix W, where each column represents the embedding vector for a factor. The embedding vector is calculated by multiplying it with the corresponding weight vector.

2.Autoencoder:
An autoencoder is a type of neural network that can learn a compact representation of data. Even though it may be challenging to implement in C language, it can be used to map continuous value factors to a lower-dimensional space

3.Rule-Based Mapping:
Encoding can be performed by rule-based mapping with binning or segmenting continuous values based on business logic or statistical characteristics, and then assign a fixed embedding vector to each bin or segment.

In comparation, Autoencoders typically require a substantial amount of data for training and may be too complex for simple embedding needs and using C. Rule-Based Mapping can be easily implemented in C language but needs specific business logic and need an expert to design the segment threshold for each factor. One-Hot coding is efficient for discrete factors but too sparse and less useful for continuous factors.

Therefore, we pick Linear Transformation as our encoding method. Furthermore, we implement based on the idea of [Zhang *et al.*, 2016] , which is to use the **Vector Weights** in 4.1 as our embedding weights. After learning these weights through training process, we can obtain the embedding for our input factors as:

Let $value_i$ be the value of the factor i, it can be discrete or continuous,

Let x_i be the embedded feature

$$x_i = i * V_i$$

where V_i is trained using previous factors using the method in 3.2.3. Our method also garuntee the dimension of V and x is the same, so they can be used to calculate dot product in equation 3.2.3.

3.2.3 The weights of Factor Models

$$y_{FM} = \langle w, x \rangle + \sum_{j_1=1}^{d} \sum_{j_2=j_1+1}^{d} \langle V_i, V_j \rangle x_{j_1} \cdot x_{j_2}$$

The output of a factorization model is denoted as `y_FM`. For a given input vector `x ∈ R^d`, the factorization model can be described by the following equation:

`y_FM = ⟨w, x⟩ + ∑(j1=1)^(d) ∑(j2=j1+1)^(d) ⟨V_i, V_j⟩ x_j1 · x_j2`

where:

- `w ∈ R^d` is a weight vector corresponding to the features in `x`.
- `V_i ∈ R^k` is a matrix where each row corresponds to the k-dimensional vector representation of the feature i.
- `⟨w, x⟩` is the dot product of `w` and `x`, reflecting the importance of first-order feature interactions.
- The double summation term represents the sum of products of the dot products of pairs of `V` vectors and the corresponding feature values in `x`, accounting for the second-order feature interactions.
- `k` is a predefined constant determining the size of the feature vector representations in `V`. In our model, we pick k as the number of our factors.

The weights of different factors can be calculated as below:

Individual features (w)

- The weights for individual features are similar to linear regression.
- These weights are learned during the training process, where the model attempts to minimize the difference between the predicted values and the actual target values (e.g., using methods like stochastic gradient descent).

Vector Weights (V):

- Each feature x_i in the feature vector **x**is associated with a vector V_i which is used to model interactions with other features.
- The length *k* of vector V_i is a hyper-parameter representing the dimensionality of the interaction space and must be chosen before training. A larger *k* allows the model to capture more complex interactions but increases computational complexity and the risk of overfitting.

  In our model, we picked k to be the total number of factors, so it at least have enough dimension to represent its relationship to all other factors.

Training Process:

- Factor weights V_i are learned from data. The model is trained to find these weights in a way that the predicted values are as close as possible to the true output values.

- The learning can be done through various optimization techniques, with gradient descent being the most common. Regularization terms are often added to the loss function to prevent overfitting.

In our model, we will use the stock daily return (today's close price - previous day's close price)/previous day's close price as our target values, and use our hardware calculated factors as inputs for training out weights using gradient descent.

Choice 1:

For simplify, we will train these weights using our hardware calculated factors and Python. Then we will set the weights on the hardware and use these weights for our FM model's calculation and embedding on software.

Choice 2:

Weight training process will be written in software ⬆

4 Resource Budgets

Our system requires storage for two distinct data segments:

1. **Historical Stock Data**: This consists of a 5-year stock record that occupies approximately 150,000 bytes, equivalent to 150 KB.
2. **Predictive Factors**: These are the computed factors necessary for prediction purposes, occupying a memory space of 5×5×8 bytes, totaling 200 bytes.

Cumulatively, our algorithm necessitates a memory capacity of 150.2 KB. This volume of data storage is well within our chip memory capacity, ensuring we remain clear of any potential memory overflow issues.

5 The Hardware/Software Interface

Data Registers:

Input Configuration:

Our data encompasses five dimensions, each requiring 8 bits, totaling 40 bits per set. To accommodate this, we utilize 64-bit registers. For the calculation of a 30-day period, we have an array of 30 such registers at our disposal. These registers are versatile, being capable of holding either the input data or the output factors.

Register 1-30:

Bits 0-7: Stores open price

Bits 8-15: Stores highest price

Bits 16-23: Stores lowest price

Bits 24-31: Stores close price

Bits 32-39: Stores volume

Output Configuration:

For the output, each day is characterized by five factors, with each factor occupying 16 bits. To store these, we employ registers that are 128 bits in size.

Register 31-60:

Bits 0-15: Stores factor 1

Bits 16-31: Stores factor 2

Bits 32-47: Stores factor 3

Bits 48-63: Stores factor 4

Bits 64-79: Stores factor 5

Control and Status Registers:

Register 61 – Algorithm Selection and Result Display:

Bits 0-2: Represents algorithm chose: 5 algorithms

Bits remain: Represents result to display

Register 62 – Calculation Completion Status:

Bits 0-2: Represents whether hardware has completed calculation