Mini Report #1

Andre Sealy, Swapnil Pant, Daiki Ishiyama

March 25, 2025

Introduction

Trend-following strategies – like moving averages or time-series momentum – are widely used in practice and have been successful across many different asset classes for decades. Despite their popularity, whether or not investors can generate excess remains questionable. Many existing trend signals are heuristic-based and chosen based on intuition or empirical trial-and-error rather than a grounded, model-based approach. This motivates our question:

With the rise of machine learning, there is an opportunity to learn price trends directly from data rather than imposing rigid parametric structures (like fixed-length moving averages). We examine price trends through the lens of supervised learning, treating the trend signal as something that can be trained to predict returns. We aim to bridge the gap between statistical modeling and practical trading signals. We will generate a framework that generalizes popular tend signals and allows for a data-driven optimal trend filter, which improves return predictability and economic outcomes.

Data & Preprocessing

Data Description

For the project, we use a comprehensive dataset that employs image classification techniques to predict stock price movements, using both structured and unstructured data, totaling 54 files organized by year. The two types of files are organized by image files (.dat) and label files, which cover 27 years from 1993 to 2019. The dataset is nearly 8 GB large, with 2,061,055 individual stock samples.

The image files (.dat) are binary files containing grayscale images of size 64x60 pixels. Each image represents a visual representation of a stock's price movement and volume history.

The images are created using a 20-day rolling window of the market data. The image files in our project will represent the features of our supervised learning algorithm.

The labels files (.feather) represent the target variable in our dataset. Each sample contains 8 columns:

- Date: Last day of 20-day rolling window,
- StockID: CRSP PERMNO identifier,
- MarketCap: Market capitalization in thousands of dollars,
- Ret_5d: 5-day forward return (primary prediction target),
- Ret_20d: 20-day forward return,
- Ret_60d: 60-day forward return,
- Ret_month: Monthly return (from current month-end to next month-end),
- EWMA_vol: Exponentially weighted volatility with alpha 0.05.

The classification task is to create a binary prediction of the target variable, 5-day returns (positive/negative examples), which transforms traditional time series forecasting into a visual pattern recognition problem.

Preprocessing

We scale up the pipeline for multiple years, systematically loading and combining data year by year. The splitting utilized for our training and validation process involves a time-series split. We have decided to use the interval from 1993 to 2014 for our training and validation set, with the remaining 5 years for testing. This will efficiently allow our model to generalize on the unseen observations of our dataset.

Our preprocessing step also involves parallel label processing. The .feather format is specifically chosen for its performance with columnar data and fast read/write operations. After processing all of the years individually, we perform two critical concatenation operations. First, we merge all yearly image arrays into a single continuous NumPy array along the first dimension. This process creates one large tensor containing all training images. We also combine all yearly label DataFrames into a single DataFrame, preserving all column structures and maintaining alignment with the image data. The final result gives us around 1.78 million samples (1,787,748 to be exact), each with shape (64, 60) for images and 8 columns of metadata for labels.

After we perform the data aggregation step, we are left with a large NumPy array containing our image data and a DataFrame with all the corresponding labels. It should also be noted that each time we load the dataset, we create a copy of the data that detaches the data from these file mappings, preventing potential issues with file handles and memory access during training. It ensures the tensor data is contiguous in memory, which improves computational efficiency.

Time Series Splitting and Validation

As mentioned previously, we use time series splitting to train the neural network, meaning we preserve the chronological ordering, which is critical for financial time series data. We take the first 70% of samples for training and the remaining 30% for validation. During the split, we transform the raw 5-day returns into binary labels, where label one represents the label if the 5-day return is positive and zero otherwise. This binary transformation is an important part of the preprocessing step, as it turns a traditional regression problem into a classification problem. (we have also created an option for random splitting, but have elected not to utilize this approach)

The final part of the preprocessing involves utilizing PyTorch's DataLoader class. Using the DataLoader class, we group samples into batches of 128, which is crucial for efficient GPU processing (a form of preprocessing that transforms individual samples into structured batches). We also implement a shuffling mechanism, which randomizes the order of samples with each epoch. Randomizing the order prevents the model from learning the order of samples and reduces overfitting by presenting the data in different sequences. This process also allows us to provide more robust gradient updates.

We have also decided to implement memory pinning, which automatically pins the fetched data to CPU memory, enabling faster transfer rates to the CUDA (Compute Unified Device Architecture) enabled GPUs. This parameter also allows us to pre-allocate memory to avoid costly data transfers during the training process. We only use this preprocessing step for GPU-based training.

Methodology

Xavier Initialization Strategy

When working with deep neural networks, we must select an appropriate starting weight to train effectively and carefully. We accomplish this by employing the Xavier uniform initialization, which allows us to configure linear (fully connected) and convolutional layers with appropriate starting values. The Xavier initialization strategy also sets weights to values that maintain appropriate signal magnitude as data flows through the network during forward and backward propagation. By carefully scaling (and randomizing) the initialization process, we prevent the neurons from becoming saturated (outputting the same value) or becoming "dead" (never activating), which would impede the learning process, thus ensuring training stability.

The Xavier uniform initialization does this by setting the weights based on a uniform distribution with carefully calculated bounds. Afterward, we scale these initial values proportionally to each neuron's input and output connections. This process benefits networks with certain activation functions, such as ReLU. This initialization accounts for the kernel dimensions and number of filters for convolutional layers, while for linear layers, it considers the input and output dimensions.

Convolutional Neural Network (CNN)

Here, we specifically outline the architectural components of the convolutional neural network used to train the model. The architecture processes 64x60 grayscale images representing stock price patterns and predicts binary outcomes (positive or negative future returns).

First, we describe the input layer, which reshapes to (-1, 1, 64, 60). The -1 in the first dimension allows us to take an arbitrary batch size, while the second, third, and fourth dimensions allows us to take the color channel, height, and width of the image. The single grayscale channel perserves the intensity patterns of the financial charts.

The input layer is followed by the 3 convolutional blocks The first convolutiona box has a single input channel (the greyscale) and 64 feature maps.

Each convolutional block utilizes parameters for the kernel size, stride, dilation, and padding. These parameters offer the following benefits to the neural network:

- **Kernel (5,3):** The kernel size fits the rectangular filter, which emphasizes the vertical patterns of the data and helps detect the short-term temporal patterns. In addition to temporal patterns, the kernel size lets us capture price formation patterns like support/resistance levels, breakouts, and trend reversals.

- **Stride (3,1):** The stride filter has a vertical stride of 3 and a horizontal stride of 1. This means the filter will jump 3 pixels downward after each application and move 1 pixel to the right, ensuring no pixel is skipped. This filter allows us to sample price levels more sparsely while densely sampling the dimension, which reduces the vertical dimensions of the output feature maps while preserving temporal resolution.

- **Dilation (2,1):** The vertical dilation (2) inserts 1 gap between kernel elements vertically, while the horizontal dilation (1) ensures that there are no gaps between kernel elements, which is standard for convolutional layers. The effect of this setup expands the vertical receptive field to cover 9 pixels (5 + 4 spaces) rather than just 5. This has the added benefit of capturing broader price movements without increasing parameter count.

- **Padding (12, 1):** The vertical padding adds 12 pixels of zero padding at the top and bottom, while the horizontal padding adds a minimum of 1 pixel padding at the left and right. The larger vertical padding compensates for kernel size, stride, and dilation, which allows us to ensure feature map dimensions work with subsequent layers.

After the convolutional layer, we immediately apply batch normalization, which allows us to stabilize and accelerate training by normalizing the activations between layers. The first convolutional box normalizes 64 features, while the second and third normalizes 128 and 256 features, respectively (the same number as the output channels for each convolutional layer). Batch normalization works by calculating the mean μ and variance σ^2 of activations across the batch and spatial dimensions. Then, it normalizes by applying the following transformation:

$$\hat{x} = \frac{(x - \mu)}{\sqrt{\sigma^2 + \epsilon}}, \tag{1}$$

where ϵ is some small constant used for numerical stability. We then apply a scale and shift parameter

$$y = \gamma \hat{x} + \beta, \tag{2}$$

where γ is the scale and β is the shift; which allows the network to learn the optimal distribution for each layer.

After each batch normalization process, we implement the Leaky Rectified Linear Unit (ReLU), a modified version of the standard ReLU activation function that allows a small gradient when the input is negative. This addresses the issue of vanishing gradients or ”dying” layers. Each leaky ReLU will have a negative slope of 0.01. An additional benefit of using the Leaky ReLU instead of the standard ReLU is retaining some information from negative signals, which could represent important financial indicators like price drops or downtrends.

In our CNN, we employ MaxPooling at the end of each convolutional block. MaxPooling is a downsampling operation that reduces the spatial dimensions of feature maps by selecting the maximum value within a defined window. In all three convolutional blocks, we implement the MaxPooling operation with the kernel size of (2,1) and stride of (2,1).

The first implementation of MaxPooling will maintain all of the temporal information while condensing the price level, while the other two implementations will have a cumulative effect.

The final layer is a fully connected layer, which acts as a bridge from the convolutional features to the classification head of the neural network. The input dimension is 46,080, which is the direct result of flatting the output from the third convolutional block. The three convolutional blocks progressively transform the 64x60 inputs through multiple operations, while the MaxPooling operation in each block reduces the vertical dimension by half, preserving the horizontal dimension. The final output of the third block contains 256 feature maps (channels) with specific spatial dimensions, resulting in precisely 46,080 values per sample.

In this layer, we have a dropout parameter, which implements a dropout rate of 50%. This prevents the network from over-relying on any specific feature extracted by the convolutional layer. We also use a linear projection, which directly maps the high-dimensional feature space to a binary classification. Since we have not implemented a hidden, fully connected layer, the network relies entirely on the convolutional feature hierarchy. This structure allows the convolutional layers to do the heavy lifting of feature extraction and applies a strong regularization framework at the classification stage to prevent overfitting.

Training and Validation

For training, we implement a complete training epoch that systematically exposes the model to all training examples, calculates prediction errors, and updates the model weights to improve performance. We process the data in batches of 128 samples (as defined by the DataLoader). The forward pass involves transferring data to the appropriate device (which will most likely be the GPU) and passing financial images through the CNN architecture. The CNN will execute the following:

- Reshaping inputs to (-1, 1, 64, 60)
- Processing through three convolutional blocks
- Flattening to 46,080 features
- Applying dropout regularization
- Projecting to 2 output classes
- Computing final probabilities with the Softmax activation function

To compare the predicted probabilities against the actual binary labels, we implement a cross-entropy loss function to quantify the prediction. For a classification of task with C classes, the cross-entropy loss is the following:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log\left(\hat{y}_i\right), \tag{3}$$

where y_i is the true label (one-hot encoded, so only one value is 1, and the rest are 0) and $\hat{y}_i$ is the predicted probability for class i (usually from a softmax layer). In addition to the cross-entropy loss function, we implement the Adam optimizer, which adapts the learning rates for each parameter. We've set a conservative learning rate of 0.00001 to prevent overshooting optimal values.

The number of batches per epoch is determined by the size of our training dataset and the batch size defined on DataLoader. Since we are dealing iwth 1,787,748 samples from 1993-2014, and 70-30 split used for training and validation, we have 1,251,424 samples. We also consider the batch size of 128. Based on these parameters, we have 9,777 batches per epoch ($1,251,424/128 \approx 9,777$). Meaning, our CNN processes approximately 9,777 batches of financial images in each complete epoch. These batches allows us to provide extensive gradient updates, allowing our model to learn many different patterns presented in the financial time series data.

For the validation process, we disable the gradient computation which significantly reduces memory usage. It also has the added benefit of speeding up the processing time, since we're only measuring performance, not updating weights. The same cross-entropy loss function is used on the validation step as in the training step.

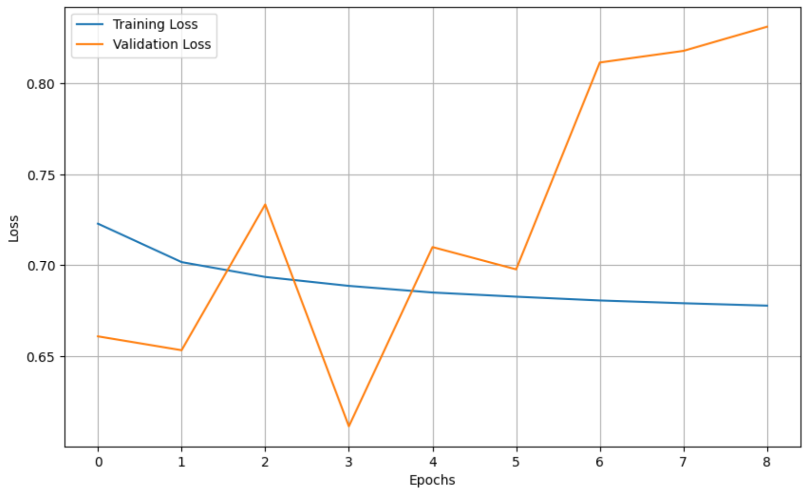

Figure 1: Training and Validation Loss

We execute each epoch's training and validation process and then capture their respective loss values, our primary quantitative performance metric. The loss values directly measure how well our CNN can predict stock movements from visual chart patterns. For each epoch, we record how our financial CNN's prediction accuracy evolves, capturing both training and validation performance. We also create a system to preserve the model states throughout training, allowing us to later select optimal versions of our financial predictors based on performance metrics.

We implement an early stop logic function to determine the optimal loss characteristics. The function identifies and records when our model performs best on validation data. It identifies the best value is identified by stopping the training and validation step when the model's performance stops improving for a specific number of epochs. This prevents wasted computation time and overfitting when the model has reached its performance plateau.