Enhancing Time Series Momentum Strategies Using Deep Neural Networks

Vinay Bharath - vab351
Xingxuan Chen - xc1494
Sachin Labhishetty - sl7466

Table of Contents

Introduction

Momentum as a risk premium in finance has been extensively documented in the academic literature, with evidence of persistent abnormal returns demonstrated across a range of asset classes, prediction horizons and time periods. In the project, we use a new method which is deep Momentum Networks – A hybrid approach to simultaneously learning both trend estimation and position sizing in an integrated data driven manner. Firstly, we generate trading signals using deep neural networks. Secondly, we optimize networks for risk-adjusted performance metric , i.e. Sharpe Ratio, by utilizing automatic differentiation in existing backpropagation frameworks. This paper in the reference might be the first to consider the use of deep learning within the context of time series momentum strategies.

Strategy Definition

The conventional method is **standard supervised learning**. **Trend Estimation**: $Y_t^{(i)} = f\left(\mathbf{u}_t^{(i)}; \boldsymbol{\theta}\right)$ where f() is the output of the machine learning model, which takes in a vector of input features u(i)t and model parameters to generate predictions. Taking volatility-normalized returns as targets, the following mean-squared error and binary cross-entropy losses can be

$$\begin{aligned}
\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}) &= \frac{1}{M}\sum_{\Omega}\left(Y_t^{(i)} - \frac{r_{t,t+1}^{(i)}}{\sigma_t^{(i)}}\right)^2 \\
\mathcal{L}_{\text{binary}}(\boldsymbol{\theta}) &= -\frac{1}{M}\sum_{\Omega}\Big\{\mathbb{I}\ \log\left(Y_t^{(i)}\right) \\
&\quad + (1-\mathbb{I})\log\left(1 - Y_t^{(i)}\right)\Big\}
\end{aligned}$$

used for training:

Position Sizing:

$$\begin{aligned}
&\text{Regression} && X_t^{(i)} = \text{sgn}(Y_t^{(i)}) \\
&\text{Classification} && X_t^{(i)} = \text{sgn}(Y_t^{(i)} - 0.5)
\end{aligned}$$

As such, we take a maximum long position when the expected returns are positive in the regression case, or when the probability of a positive return is greater than 0.5 in the classification case.

Then, we will use a new approach which is **Deep Momentum Network** to predict.
Direct Outputs:

$$X_t^{(i)} = f\left(\mathbf{u}_t^{(i)}; \boldsymbol{\theta}\right)$$

This approach generate positions directly and simultaneously learning both trend estimation and position sizing. We focus on optimizing the average return or the annualized Sharpe ratio via the loss functions below:

$$\begin{aligned}
\mathcal{L}_{\text{returns}}(\boldsymbol{\theta}) = & \quad -\mu_R \\
= & \quad -\frac{1}{M}\sum_{\Omega} R(i,t) \\
= & \quad -\frac{1}{M}\sum_{\Omega} X_t^{(i)} \ \frac{\sigma_{\text{tgt}}}{\sigma_t^{(i)}} \ r_{t,t+1}^{(i)}
\end{aligned}$$

$$\mathcal{L}_{\text{sharpe}}(\boldsymbol{\theta}) = -\frac{\mu_R \times \sqrt{252}}{\sqrt{\left(\sum_{\Omega} R(i,t)^2\right)/M - \mu_R^2}}$$

The algorithm used below is LSTM. The structure of LSTM is as follows:

$$\begin{aligned}
\mathbf{f}_t^{(i)} &= \sigma(\mathbf{W}_f \mathbf{u}_t^{(i)} + \mathbf{V}_f \mathbf{h}_{t-1}^{(i)} + \mathbf{b}_f) \\
\mathbf{i}_t^{(i)} &= \sigma(\mathbf{W}_i \mathbf{u}_t^{(i)} + \mathbf{V}_i \mathbf{h}_{t-1}^{(i)} + \mathbf{b}_i) \\
\mathbf{o}_t^{(i)} &= \sigma(\mathbf{W}_o \mathbf{u}_t^{(i)} + \mathbf{V}_o \mathbf{h}_{t-1}^{(i)} + \mathbf{b}_o) \\
\mathbf{c}_t^{(i)} &= \mathbf{f}_t^{(i)} \odot \mathbf{c}_{t-1}^{(i)} \\
&\quad + \mathbf{i}_t^{(i)} \odot \tanh(\mathbf{W}_c \mathbf{u}_t^{(i)} + \mathbf{V}_c \mathbf{h}_{t-1}^{(i)} + \mathbf{b}_c) \\
\mathbf{h}_t^{(i)} &= \mathbf{o}_t^{(i)} \odot \tanh(\mathbf{c}_t^{(i)}) \\
Z_t^{(i)} &= g\left(\mathbf{W}_z \mathbf{h}_t^{(i)} + \mathbf{b}_z\right),
\end{aligned}$$

Deep Momentum Network

LSTM are particularly well suited to predict time series with lags of unknown duration. This relative insensitivity to to gap length gives it an advantage compared to other methods like RNN's and other sequence learning methods. Here we use two layers of LSTMs followed by a dense layer consisting of the number of assets in consideration. Dropout layers are added between each successive LSTM layer. A lookback period of 10 days was used along with a batch size of 22 which roughly corresponds to a month of data per batch.

Network Architecture

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 10, 50)            16800
_________________________________________________________________
dropout_1 (Dropout)          (None, 10, 50)            0
_________________________________________________________________
lstm_2 (LSTM)                (None, 40)                14560
_________________________________________________________________
dropout_2 (Dropout)          (None, 40)                0
_________________________________________________________________
dense_1 (Dense)              (None, 25)                1025
_________________________________________________________________
dropout_3 (Dropout)          (None, 25)                0
_________________________________________________________________
dense_2 (Dense)              (None, 11)                286
=================================================================
Total params: 32,671
Trainable params: 32,671
Non-trainable params: 0
```

Custom loss

As opposed to using conventional regression and classification loss such as mean squared error or cross entropy, a new technique of directly optimizing for risk adjusted performance metric such as Sharpe ratio was tested out. The paper proposed used variance of individual daily returns for computing Sharpe ratio, but we also implemented a method to use daily portfolio variance in the Sharpe ratio.

```python
import tensorflow as tf
import keras.backend as K

def compute_sharpe(captured_returns):

    ave_ret = tf.reduce_mean(captured_returns)
    ave_sq_rets = tf.reduce_mean(tf.square(captured_returns))
    variance = ave_sq_rets - tf.square(ave_ret)
    std = tf.sqrt(variance + 1e-9)

    sharpe = ave_ret / std * tf.sqrt(252.0)
    return sharpe


def sharpe_ratio_loss(y_true, y_pred):

    weights = y_pred
    captured_returns = weights * y_true

    sharpe = compute_sharpe(captured_returns)

    return -sharpe

def compute_sharpe2(captured_returns):
    """Function that calculates daily returns first and then calcualtes sharpe"""

    ave_ret = tf.reduce_mean(captured_returns)
    daily_ret = tf.reduce_mean(captured_returns, axis=1)
    #ave_sq_rets = tf.reduce_mean(tf.square(captured_returns))
    #variance = ave_sq_rets - tf.square(ave_ret)
    #std = tf.sqrt(variance + 1e-9)

    sharpe = ave_ret / tf.math.reduce_std(daily_ret) * tf.sqrt(252.0)
    return sharpe

def sharpe_ratio_loss2(y_true, y_pred):

    weights = y_pred
    captured_returns = weights * y_true

    sharpe = compute_sharpe2(captured_returns)

    return -sharpe
```

Performance Evaluation:

The network generates an output of weights to be allocated for each of the input ETF indices. The weights are generated straight out the network are from a range of -1 to 1. A positive weight indicates to go long by the corresponding amount and a negative weight indicates to go short. The positions are taken on a daily basis, adjusting the PnL with the same frequency. We assume there are no transaction costs in the computation of the PnL.

Calculation of the PnL:

The calculation was carried out in three steps:

1) Normalising the ndarray of daily weights generated by the network.

```
[ ] def custom_normalise(x):
        return x/sum(abs(x))
```

2) Computed the lagged daily percentage change for all assets in the portfolio.

```
df_returns1 = df_prices.pct_change(1).add_suffix('_1day')
```

3) Perform an element-wise ndarray multiplication for net portfolio PnL on a daily basis. Find the cumulative returns over time by using cumprod, as the results are in %age terms.

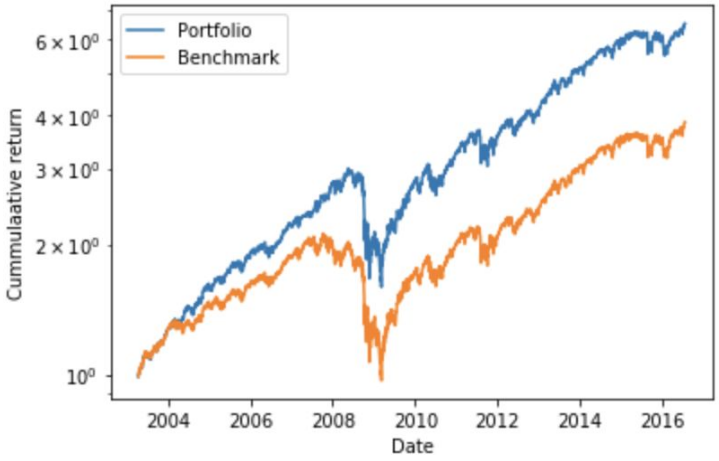
```
#df_portfolio_returns = df_daily_returns*normalize(df_train_weights, axis=1, norm='l1')
df_portfolio_returns = df_daily_returns*np.apply_along_axis(custom_normalise,1,df_train_weights)
df_portfolio_returns = np.sum(df_portfolio_returns, axis=1)
df_cumulative_pr = np.cumprod(1+df_portfolio_returns)
```

Comparison of test & train results with the benchmark portfolio:

Traditional momentum models predicted the direction of the instrument without any suggestions on the weightage it should be given when taking a position. To calculate the returns for the benchmark model, we go long on all instruments and assign equal weightage to all of them. We used cumprod to find the cumulative returns over training and test period. The plots for training and test data are plotted below:

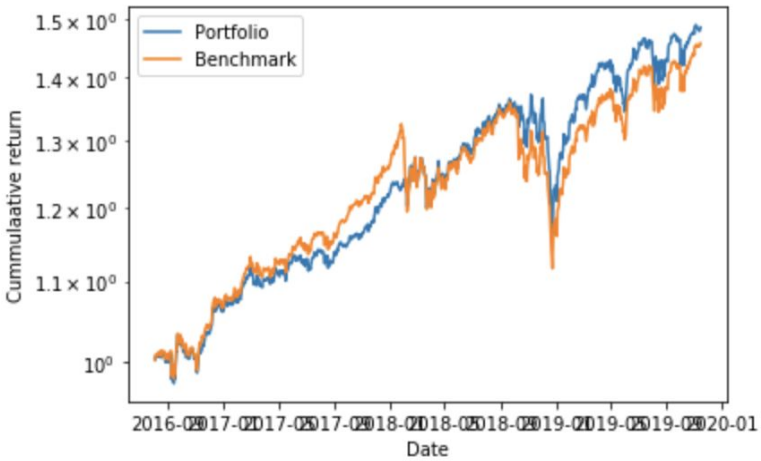
Training results:

The cumulative returns for the training set was ~600% for 12 years, which simplifies to a compounded 16% annual returns. The benchmark at the same period accumulated ~300% cumulative returns. The resulting portfolio sharpe was 0.91 as opposed to 0.63 for the benchmark



Test results:

The network performance resulted in a 14.47% annual returns for the test data from 2016 to 2019, which accumulated to 50% over three years. As for the benchmark, the annual returns was 11.8% and compounded to 40%. The out of sample sharpe for the portfolio was 1.22 as opposed to 1.06 for the benchmark.



Conclusions

Trading rules were directly generated by the Deep momentum network by optimizing for risk adjusted performance metric like sharpe ratio. Though the strategy ignores some practical aspects of trading such as transaction costs, it can be seen that this hybrid method for learning both trend estimation and position sizing directly from the data results in better Sharpe out of sample.

Reference

[1] Lim, Bryan and Zohren, Stefan and Roberts, Stephen, “Enhancing Time Series Momentum Strategies Using Deep Neural Networks (April 9, 2019),” *The Journal of Financial Data Science*, Fall 2019. Available at SSRN: https://ssrn.com/abstract=3369195 or http://dx.doi.org/10.2139/ssrn.3369195