

# Capstone Project Report for Training a Stock Price Prediction Model with Neural Network

Jianming Pan

August 2022

## Definition

### 1 Project Overview

Stock price prediction is always a challenging but attracting task. Various techniques have been tried to conquer the market, but few succeed. The key component for stock price prediction task is to fit a conditional distribution function of stock returns in the context of historical data. And then input updated information to make a prediction of stock's return and risk. Based on estimated return and risk, we also need to utilize portfolio optimization to allocate resources on making investment decisions.

The difficulty of such tasks mostly lies in the low signal-to-noise-ratio (SNR) of stock market. Day-by-day transaction in the busy market generates huge amount of data, which is perfect for modern machine learning method. However, it usually takes great effort to extract the valuable part of such data.

In this project, I created a Multi-Precision-Layer (MLP) model with a state-of-art technique, the DPP optimization layer<sup>1</sup>, for stock price prediction. The trained model uses Microsoft open source qlib Alpha158 dataset<sup>2</sup>.

### 2 Problem Statement

As mentioned before, our major task is to train a model(function)  $F : R^{n \times m} \rightarrow R^n$ , mapping  $n$  stocks with  $m$  features to their returns. To be clear, predicting stock price is the same as to predict stock return. Because we could choose a start time and set the stock price as  $P_0$ , and then at any time later than the start time. The price of stock is  $P_0(1 + r)$ , where  $r$  represents the return in the interval time. By this transformation, our data distributed more like independently.

---

<sup>1</sup>Agrawal A, Amos B, Barratt S, et al. Differentiable convex optimization layers[J]. Advances in neural information processing systems, 2019, 32.

<sup>2</sup><https://github.com/microsoft/qlib.git>

More specifically, we train to minimize the prediction loss  $L_0(\theta)$  in the context of  $x, y \sim \mathcal{D}$

$$\underset{\theta}{\text{minimize}} \quad L_0(\theta) = \mathbf{E}_{x, y \sim \mathcal{D}} [(F(x; \theta) - y)^2] \quad (1)$$

Then we need to allocate resources to make decision  $z$  minimizing the task loss  $L_1(z)$  using predictions from the first step

$$z^*(x; \theta) = \underset{z}{\text{argmin}} L_1(z) = \mathbf{E}_{y \sim p(y|x; \theta)} [g(x, y, z)] \quad (2)$$

### 3 Metrics

For all investment decision, backtest is the best way of determining which model performs better. We will make a backtest program to simulate the true investment environment. As for the metrics, annualized return (AR), sharpe ratio (SR), IC, IR, Maxdrawdown, and net value of simulated portfolio is all we need.

$$AR = \sqrt[T/252]{\prod_{i=1}^T (1 + r)^i} \quad (3)$$

$$SR = \frac{AR}{\sigma(r)} \quad (4)$$

$$IR = SR = \frac{AR - benchmark}{\sigma(r)} \quad (5)$$

$$Maxdrawdown = \max_{i < j} (r_i - r_j) \quad (6)$$

## Analysis

### 4 Data Exploration

qlib contains tons of stock transaction data. However, in this project, we only use Alpha158 of China's stock market. Users could use git clone command download qlib from <https://github.com/microsoft/qlib.git>. Then open the terminal window in the qlib file with command `scripts/get_data.py qlib_data -target_dir ./qlib/qlib_data/cn_data -region cn`.

Alpha158 has two major indexes: "feature" and "label". "feature" has 158 types of stock cross-sectional feature made of basic Open/High/Low/Close and volume data. "label" is next day's return.

Figure 1 shows that "feature" data is a pandas DataFrame with zipped indexes (datetime, instrument). The column of "feature" is 158 features of each stock.

Figure 2 shows that "label" is next days' return.

Figure 3 shows the data configuration

|            |            | KMID      | KLEN      | KMID2     | KLOW      | KLOW2     | \ |
|------------|------------|-----------|-----------|-----------|-----------|-----------|---|
| datetime   | instrument |           |           |           |           |           |   |
| 2010-01-04 | SH600000   | -0.916175 | 0.616999  | -0.719628 | -0.343883 | -0.447256 |   |
|            | SH600009   | 3.236425  | 2.022583  | 3.458775  | -0.720924 | -0.736838 |   |
|            | SH600010   | 0.158732  | -0.998949 | -0.643470 | -0.720924 | -0.736838 |   |
|            | SH600011   | 0.363964  | -0.993337 | -0.185469 | -0.044151 | 0.242481  |   |
|            | SH600015   | 0.503794  | -0.094025 | 0.693795  | -0.057317 | -0.093970 |   |

Figure 1: Sample of feature

|            |            | LABEL0    |
|------------|------------|-----------|
| datetime   | instrument |           |
| 2010-01-04 | SH600000   | -1.007970 |
|            | SH600009   | -0.885182 |
|            | SH600010   | 0.394897  |
|            | SH600011   | -0.142323 |
|            | SH600015   | -2.090131 |

Figure 2: Sample of label

```
dataset_config = {
    "class": "DatasetH",
    "module_path": "qlib.data.dataset",
    "kwargs": {
        "handler": {
            "class": "Alpha158", # Using Alpha158 as features
            "module_path": "qlib.contrib.data.handler",
        },
        "segments": {
            "train": ("2010-01-01", "2014-12-31"),
            "valid": ("2015-01-01", "2016-12-31"),
            "test": ("2017-01-01", "2020-06-01")
        }
    }
}
```

Figure 3: Data configuration

## 5 Exploratory Visualization

The plot below shows how stock returns distributed across sample space. Return data was processed by cross-sectional normalization. So you can see in the plot that all returns are in interval  $[-10, +10]$ . More information about data processing would be displayed on Data Processing section.

Figure 5 shows that the whole market changes a lot in the test time, which requires our model to have a good generalization ability.

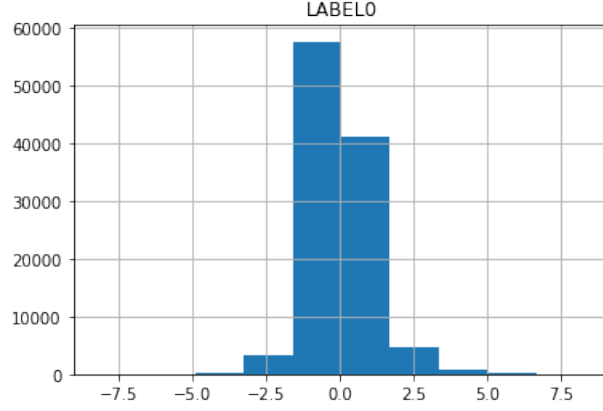


Figure 4: Return distribution. The return distributed symmetric with 0. The plot shows that negative returns happened more in the history. Also, the return is slightly inclined to left side

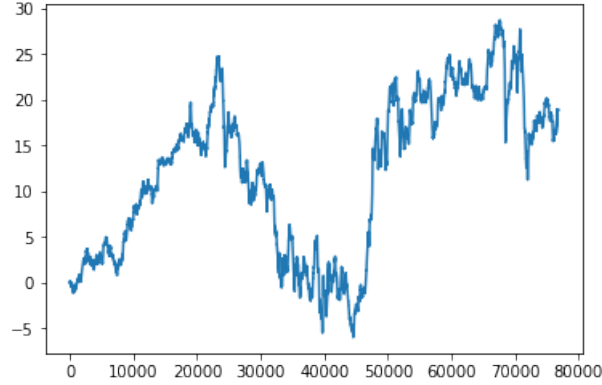


Figure 5: Overview of how stock market changes across sample time interval. The market experienced great depression and great rise up in this interval, which really causes challenges for our model

## 6 Algorithms and Techniques

MLP is a universal function approximator<sup>3</sup>. Zhang, et al.(2021) shows that MLP with one hidden has a good approximation ability in the task of stock price prediction. We follow this paradigm and uses a MLP model with one hidden layer.

Real-time investment decision also requires a step of optimization. We uses

---

<sup>3</sup>Zhang C, Zhang Z, Cucuringu M, et al. A universal end-to-end approach to portfolio optimization via deep learning[J]. arXiv preprint arXiv:2111.09170, 2021.

DPP to include this optimization as layers of MLP network. So all the parameters of our model could be trained by SGD. Here is the loss function that includes a optimization layer.

$$\underset{\theta}{\text{minimize}} \quad L(\theta) = \mathbf{E}_{x,y \sim \mathcal{D}} [f(x, y, z^*(x; \theta))] \quad (7)$$

To train such model, we need to compute

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial z^*} \frac{\partial z^*}{\partial \theta} \quad (8)$$

Jacobian  $\frac{\partial z^*}{\partial \theta}$  is not easily attainable for general cases. But for **Convex Optimization**, KKT optimal conditions and the implicit function theorem ensure us to calculate Jacobian  $\frac{\partial z^*}{\partial \theta}$  nearly without cost.

Luckily, portfolio optimization often presents as a convex optimization of risk-adjusted return. For example, Mean-Variance Optimization (MVO) is a common paradigm in portfolio optimization. MVO features a trade-off between returns and risk.

$$\begin{aligned} z^* = \underset{z}{\operatorname{argmax}} \quad & \alpha^T z - \frac{1}{2} \lambda z^T \Sigma z \\ \text{subject to} \quad & 1^T z = 1 \\ & z \succeq 0 \end{aligned} \quad (9)$$

$z^*, \alpha, \Sigma, \lambda$ : weight, estimated returns, estimated covariance matrix, risk-aversion coefficient.

Follow this paradigm, we uses CvxpyLayers<sup>4</sup> to presents our portfolio optimization problem as convex optimization layers in our model.

The parameters of our model which can be tuned are as follows:

- learning rate
- batch size
- epoch
- hidden layer parameters
- data process

I used a Macbook pro, which does not support cuda. So the epoch set for the Optimization MLP is 10 due to computation limitation. Also, optimization relies heavily on CPU to run forward output. The overall performance of the program may be strongly constrained by users' computer configuration.

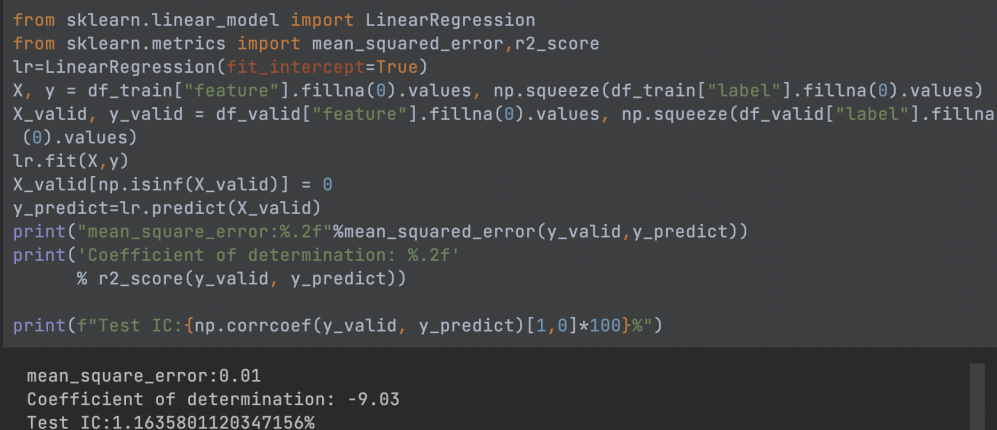
---

<sup>4</sup><https://github.com/cvxgrp/cvxpylayers.git>

## 7 Benchmark

Traditional stock price prediction task uses a linear model to integral all stock features. So the benchmark of our model is the linear regression model. We expected that MLP model outperformed Linear model. And also, since there is another optimization MLP model to be trained, the second benchmark is the previous MLP model. We also expected MLP model with optimization layers to outperform MLP model.

Linear model code is as follows:



```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
lr=LinearRegression(fit_intercept=True)
X, y = df_train["feature"].fillna(0).values, np.squeeze(df_train["label"].fillna(0).values)
X_valid, y_valid = df_valid["feature"].fillna(0).values, np.squeeze(df_valid["label"].fillna(0).values)
lr.fit(X,y)
X_valid[np.isinf(X_valid)] = 0
y_predict=lr.predict(X_valid)
print("mean_square_error: %.2f"%mean_squared_error(y_valid,y_predict))
print('Coefficient of determination: %.2f'
      % r2_score(y_valid, y_predict))

print(f"Test IC: {np.corrcoef(y_valid, y_predict)[1,0]*100}%")

mean_square_error:0.01
Coefficient of determination: -9.03
Test IC:1.1635801120347156%
```

Figure 6: Screenshot of Linear model

The Test IC in valid set is 1.16%, we expect Machine Learning model to exceed this. However, higher IC does not mean the model is better, the ultimate evaluation is backtest.

## Methodology

### 8 Data Prepossessing

qlib provides abundant data prepossessing methods when import Alpha158 dataset. For stabilizing the input of our data within each batch, we will choose cross-sectional normalization, which means all stock will be Z-scored for each of the trading day.(Only stocks traded on the same day will be included as normalization, the type of data to be normalized is cross-sectional, so it is called cross-sectional normalization). Also, it is clear that we could not do cross-sectional normalization of returns when doing inference. Because that would be using future data. So we separate data processing stage as learning stage and inference stage.

We also implemented drop NAN data method. However, I found that there are five features in Alpha158 are the same number in certain days, which makes the CS-norm failed with infinity numbers. So we simply drop them.

dataset\_config.json file is my data preprocessing configuration.

```
market = "csi100"
benchmark = "SH000903"
data_handler_config = {
    "start_time": "2008-01-01",
    "end_time": "2020-06-01",
    "fit_start_time": "2008-01-01",
    "fit_end_time": "2014-12-31",
    "instruments": market,
    "learn_processors": [
        {
            "class": "DropCol",
            "kwargs": {"col_list": ["VWAP0", "KUP", "KUP2", "HIGH0", "IMIN5"]}
        },
        {
            "class": "DropnaProcessor",
            "kwargs": {"fields_group": "feature"}
        },
        {
            "class": "DropnaProcessor",
            "kwargs": {"fields_group": "label"}
        },
        {
            "class": "CSZScoreNorm",
            "kwargs": {"fields_group": "label"}
        },
        {
            "class": "CSZScoreNorm",
            "kwargs": {"fields_group": "feature"}
        }
    ]
}
```

Figure 7: Snapshot of Alpha158 data configuration in learning stage

During inference stage, the configuration is as follows:

```

"infer_processors": [
  {
    "class" : "DropCol",
    "kwargs":{"col_list": ["VWAP0", "KUP", "KUP2", "HIGH0", "IMIN5"]}
  },
  {
    "class" : "DroonaProcessor",
    "kwargs":{"fields_group": "feature"}
  },
  {
    "class" : "DroonaProcessor",
    "kwargs":{"fields_group": "label"}
  },
  {
    "class": "CSZScoreNorm",
    "kwargs": {"fields_group": "feature"}
  }
]

```

Figure 8: Snapshot of Alpha158 data configuration in inference stage

## 9 Implementation

The code was done in a jupyter notebook(capstone.ipynb) with detailed description of each step.

The aim of this project is to train a model that helps investor make decisions on the stock market. By applying Machine Learning method, we could outperform the traditional method. To brief summary, the project is designed as follows:

(1) Load the Alpha158 from qlib. All data imported as time-series data with pandas DataFrame type. Spilt train set, valid set and test set.

(2) Built data prepossessing configuration to process data before training. Make sure the process procedure is different in stage of learning and inference. Because we do not want to introduce future data in the inference stage, while in the learning stage, normalizing all distribution in each batches requires us to mix data together. Also, cross-section norm and NAN data should be included in this script.

(3) Determine the hyperparameters of the MLP model. Initially, I chose batch size=128, learning rate=0.001, epoch=50. The net structure of the model is simple but powerful. (More trails of model hyperparameters are included in model.save file. Please note that those models are not listed in jupyter notebook or report)

(4) Predict future information with rolling data in the test stage. And using updated return to make investment decision by portfolio optimization (9).

(5) Implement backtest to calculate metrics listed above. qlib has a well-designed backtest system. However, it does not suit our model output. So I inherit Strategy class of qlib and override the generate\_target\_weight\_position()



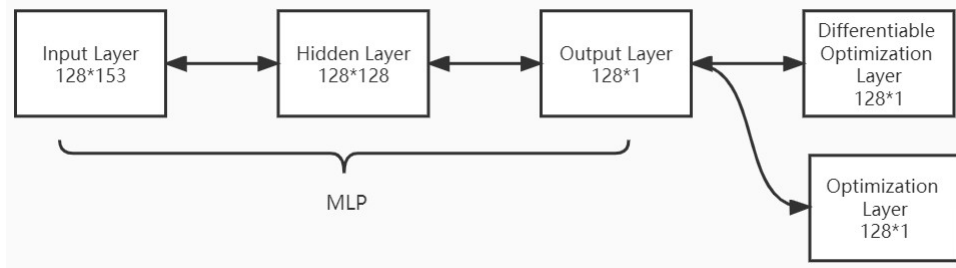


Figure 9: Net structure of the model with batch size of 128

method.

## 10 Refinement

As mentioned before, I implemented an optimization layer to refine the old predict-and-then-optimize paradigm. The net structure of refined model is Figure 9. Here I presents a more detailed optimization layer.

```

def SPOLayers(n):
    wtilde = cp.Variable(n)
    f = cp.Variable(n)

    alpha = cp.Parameter(n)
    S = cp.Parameter((n, n))
    gamma = cp.Parameter(1, nonneg=True)

    expected_return = alpha @ wtilde
    expected_risk = cp.sum_squares(f)

    constraints = [cp.sum(wtilde) == 1, wtilde >= 0, cp.norm(wtilde, "inf") <= .5, f == S @ wtilde]

    prob = cp.Problem(
        cp.Maximize(expected_return - gamma * expected_risk),
        constraints)
    return CvxpyLayer(prob, [alpha, S, gamma], [wtilde, f], gp = False)

n = 128
trading_policy = SPOLayers(n)

```

Figure 10: Optimization layer code

The optimization problem is designed to be a convex problem. Where I set the max weight of each stock must greater than 0 and less than 0.5 (To avoid holding concentration). And to match the 128 batch size, here the SPOLayers maximize 128 assets.

## Result

### 11 Model Evaluation and Validation

During development, a valid set was used to evaluate the model. I also trained alternative model with learning rate = 0.0001.(model saved in model.save file) Here the result only shows the best one of them: (learning rate = 0.001)

In evaluation stage, we do not draw the loss to evaluate our model because it is of no finance meaning. Basically, we often to choose MSE as the loss function, while we evaluate them using IC/RankIC, which is the correlation of expected returns and real returns. Because stock data distributed randomly and follows a process called random walk. If you choose the MSE as evaluation, you may find that MSE changes greatly in train set and valid/test set. So for convenience, we change to IC as the evaluation index.

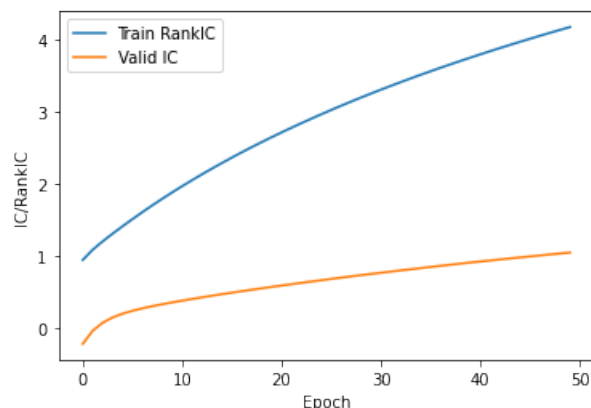


Figure 11: MLP model Train and Test IC

The plot shows that Valid IC increases lower than Train RankIC, and 50 epochs seems a good fit for this training. Also, the Test IC in valid set of MLP model is 1.05%, which is actually lower than Linear model. However, as mentioned before, IC does not tell us which model is better. It is just a signal that how well our model fitted with the data. We need to compare them in test set with backtesting metrics.

For Optimization MLP model, it has a higher IC (2.90%). Also, from Figure 12, we could see that the valid IC actually increases at first and then drop a little at epoch 5-6. The final IC of Optimization MLP model is quite stable with 10 epochs.

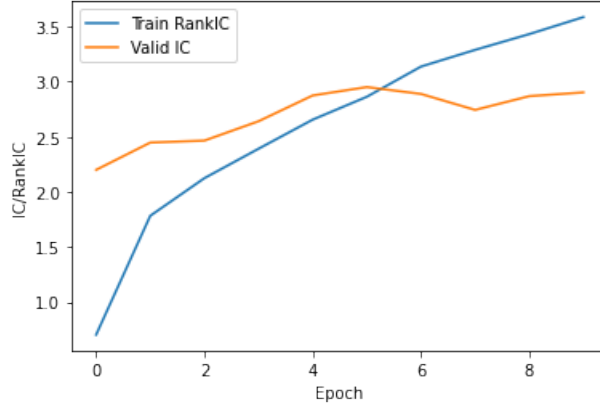


Figure 12: Optimization MLP model Train and Test IC

## 12 Justification

In this section I did backtesting for 3 models. Here is the result. SH000903 is the market index of the same period and the same stock pool as the other models. It is common to test model's return subtract the index's return. As showed below, Optimization MLP model performs best in AR, IR/SR. And MLP performs best at Max drawdown. Linear model performs best at std.

| name              | Linear model    | MLP model       | Opt model       | SH000903  |
|-------------------|-----------------|-----------------|-----------------|-----------|
| mean              | -0.000208       | -0.000078       | <b>0.000021</b> | 0.000383  |
| std               | <b>0.003113</b> | 0.003158        | 0.006127        | 0.012031  |
| annualized_return | -0.049584       | -0.018501       | <b>0.005014</b> | 0.091168  |
| information_ratio | -1.032555       | -0.379776       | <b>0.05304</b>  | 0.491206  |
| max_drawdown      | -0.17253        | <b>-0.10017</b> | -0.126793       | -0.347247 |

From Figure 13, we can have a clear look of how these three models evolve through time. Optimization MLP model outperforms the other 2 models. But in the year of 2018, MLP model and Optimization MLP model seems perform the same, which may suggest optimization layer does not work in great depression.

## Conclusion

### 13 Existing problem

Computational limitation prevents further exploration of hyper-parameters. The overall running time of 10 epoch of the last model is approximately 50 minutes.

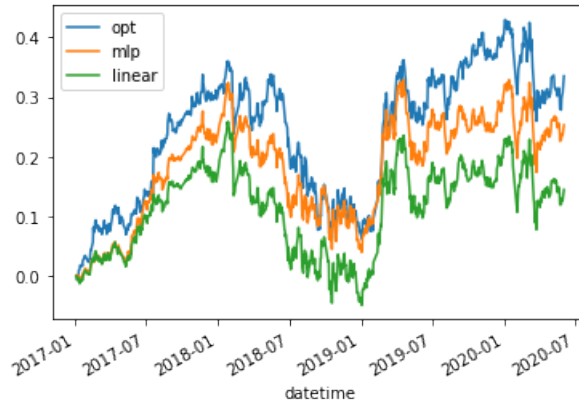


Figure 13: 3 models net value in test set

Also, in this project, we ignore the transaction cost, which is unrealistic for real-time stock investing.

## 14 Reflection

In this project, the process used could be summarized as follows:

- (1) Load stock data from qlib
- (2) Using qlib data preprocessing configuration to transform date.
- (3) Building model structure using pytorch.
- (4) Construct train, test and predict function for each model.
- (5) Inherit qlib Strategy Class to build my own backtest interface and finally implement backtesting.

Among those steps, the most challenging one is to build model structure, which I found many resources on Udacity lessons.

## 15 Improvement

At least 3 points could be improved in this project. (1) Train more models to determine the better hyperparameters for each model. (2) Expand the stock pool to other market. (3) Utilize deeper net for training.

## 16 References

- [1] <https://github.com/microsoft/qlib.git>
- [2] Agrawal A, Amos B, Barratt S, et al. Differentiable convex optimization layers[J]. Advances in neural information processing systems, 2019, 32.