05_cnn_for_trading_feature_engineering

September 29, 2021

1 CNN for Trading - Part 1: Feature Engineering

To exploit the grid-like structure of time-series data, we can use CNN architectures for univariate and multivariate time series. In the latter case, we consider different time series as channels, similar to the different color signals.

An alternative approach converts a time series of alpha factors into a two-dimensional format to leverage the ability of CNNs to detect local patterns. Sezer and Ozbayoglu (2018) propose CNN-TA, which computes 15 technical indicators for different intervals and uses hierarchical clustering (see Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning) to locate indicators that behave similarly close to each other in a two-dimensional grid.

The authors train a CNN similar to the CIFAR-10 example we used earlier to predict whether to buy, hold, or sell an asset on a given day. They compare the CNN performance to "buy-and-hold" and other models and find that it outperforms all alternatives using daily price series for Dow 30 stocks and the nine most-traded ETFs over the 2007-2017 time period.

The section on *CNN for Trading* consists of three notebooks that experiment with this approach using daily US equity price data. They demonstrate 1. How to compute relevant financial features 2. How to convert a similar set of indicators into image format and cluster them by similarity 3. How to train a CNN to predict daily returns and evaluate a simple long-short strategy based on the resulting signals.

1.1 Creating technical indicators at different intervals

We first select a universe of the 500 most-traded US stocks from the Quandl Wiki dataset by dollar volume for rolling five-year periods for 2007-2017.

- Our features consist of 15 technical indicators and risk factors that we compute for 15 different intervals and then arrange them in a 15x15 grid.
- For each indicator, we vary the time period from 6 to 20 to obtain 15 distinct measurements.

1.2 Imports & Settings

To install `talib` with Python 3.7 follow these instructions.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
from talib import (RSI, BBANDS, MACD,
                   NATR, WILLR, WMA,
                   EMA, SMA, CCI, CMO,
                   MACD, PPO, ROC,
                   ADOSC, ADX, MOM)
import seaborn as sns
import matplotlib.pyplot as plt
from statsmodels.regression.rolling import RollingOLS
import statsmodels.api as sm
import pandas_datareader.data as web
import pandas as pd
import numpy as np
from pathlib import Path
%matplotlib inline
```

[3]:
```
DATA_STORE = '../data/assets.h5'
```

[4]:
```
MONTH = 21
YEAR = 12 * MONTH
```

[5]:
```
START = '2000-01-01'
END = '2017-12-31'
```

[6]:
```
sns.set_style('whitegrid')
idx = pd.IndexSlice
```

[7]:
```
T = [1, 5, 10, 21, 42, 63]
```

[8]:
```
results_path = Path('results', 'cnn_for_trading')
if not results_path.exists():
    results_path.mkdir(parents=True)
```

1.3 Loading Quandl Wiki Stock Prices & Meta Data

[9]:
```
adj_ohlcv = ['adj_open', 'adj_close', 'adj_low', 'adj_high', 'adj_volume']
```

[10]:
```
with pd.HDFStore(DATA_STORE) as store:
    prices = (store['quandl/wiki/prices']
              .loc[idx[START:END, :], adj_ohlcv]
              .rename(columns=lambda x: x.replace('adj_', ''))
              .swaplevel()
              .sort_index()
             .dropna())
    metadata = (store['us_equities/stocks'].loc[:, ['marketcap', 'sector']])
ohlcv = prices.columns.tolist()
```

[11]:
```
prices.volume /= 1e3
prices.index.names = ['symbol', 'date']
metadata.index.name = 'symbol'
```

1.4 Rolling universe: pick 500 most-traded stocks

[12]:
```
dollar_vol = prices.close.mul(prices.volume).unstack('symbol').sort_index()
```

[13]:
```
years = sorted(np.unique([d.year for d in prices.index.get_level_values('date').
 ↪unique()]))
```

[14]:
```
train_window = 5 # years
universe_size = 500
```

[15]:
```
universe = []
for i, year in enumerate(years[5:], 5):
    start = str(years[i-5])
    end = str(years[i])
    most_traded = (dollar_vol.loc[start:end, :]
                   .dropna(thresh=1000, axis=1)
                   .median()
                   .nlargest(universe_size)
                   .index)
    universe.append(prices.loc[idx[most_traded, start:end], :])
universe = pd.concat(universe)
```

[16]:
```
universe = universe.loc[~universe.index.duplicated()]
```

[17]:
```
universe.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 2530228 entries, ('BRK_A', Timestamp('2000-01-03 00:00:00')) to
('BLL', Timestamp('2017-12-29 00:00:00'))
Data columns (total 5 columns):
 #   Column  Non-Null Count    Dtype
---  ------  --------------    -----
 0   open    2530228 non-null  float64
 1   close   2530228 non-null  float64
 2   low     2530228 non-null  float64
 3   high    2530228 non-null  float64
 4   volume  2530228 non-null  float64
dtypes: float64(5)
memory usage: 106.4+ MB
```

[18]:
```
universe.groupby('symbol').size().describe()
```

[18]:
```
count     735.000000
mean     3442.487075
std      1145.365643
min      1043.000000
25%      2368.000000
50%      3792.000000
75%      4527.000000
max      4528.000000
dtype: float64
```

[19]:
```
universe.to_hdf('data.h5', 'universe')
```

1.5 Generate Technical Indicators Factors

[20]:
```
T = list(range(6, 21))
```

1.5.1 Relative Strength Index

[21]:
```
for t in T:
    universe[f'{t:02}_RSI'] = universe.groupby(level='symbol').close.apply(RSI,␣
 ↪timeperiod=t)
```

1.5.2 Williams %R

[22]:
```
for t in T:
    universe[f'{t:02}_WILLR'] = (universe.groupby(level='symbol',␣
 ↪group_keys=False)
     .apply(lambda x: WILLR(x.high, x.low, x.close, timeperiod=t)))
```

1.5.3 Compute Bollinger Bands

[23]:
```
def compute_bb(close, timeperiod):
    high, mid, low = BBANDS(close, timeperiod=timeperiod)
    return pd.DataFrame({f'{timeperiod:02}_BBH': high, f'{timeperiod:02}_BBL':␣
 ↪low}, index=close.index)
```

[24]:
```
for t in T:
    bbh, bbl = f'{t:02}_BBH', f'{t:02}_BBL'
    universe = (universe.join(
        universe.groupby(level='symbol').close.apply(compute_bb,
                                                      timeperiod=t)))
    universe[bbh] = universe[bbh].sub(universe.close).div(universe[bbh]).
 ↪apply(np.log1p)
    universe[bbl] = universe.close.sub(universe[bbl]).div(universe.close).
 ↪apply(np.log1p)
```

1.5.4 Normalized Average True Range

[25]:
```
for t in T:
    universe[f'{t:02}_NATR'] = universe.groupby(level='symbol',
                                group_keys=False).apply(lambda x:
                                                NATR(x.high, x.low, x.
 ↪close, timeperiod=t))
```

1.5.5 Percentage Price Oscillator

[26]:
```
for t in T:
    universe[f'{t:02}_PPO'] = universe.groupby(level='symbol').close.apply(PPO,␣
 ↪fastperiod=t, matype=1)
```

1.5.6 Moving Average Convergence/Divergence

[27]:
```
def compute_macd(close, signalperiod):
    macd = MACD(close, signalperiod=signalperiod)[0]
    return (macd - np.mean(macd))/np.std(macd)
```

[28]:
```
for t in T:
    universe[f'{t:02}_MACD'] = (universe
                  .groupby('symbol', group_keys=False)
                  .close
                  .apply(compute_macd, signalperiod=t))
```

1.5.7 Momentum

[29]:
```
for t in T:
    universe[f'{t:02}_MOM'] = universe.groupby(level='symbol').close.apply(MOM,␣
 ↪timeperiod=t)
```

1.5.8 Weighted Moving Average

[30]:
```
for t in T:
    universe[f'{t:02}_WMA'] = universe.groupby(level='symbol').close.apply(WMA,␣
 ↪timeperiod=t)
```

1.5.9 Exponential Moving Average

[31]:
```
for t in T:
    universe[f'{t:02}_EMA'] = universe.groupby(level='symbol').close.apply(EMA,␣
 ↪timeperiod=t)
```

1.5.10 Commodity Channel Index

[32]:
```
for t in T:
    universe[f'{t:02}_CCI'] = (universe.groupby(level='symbol',␣
↪group_keys=False)
     .apply(lambda x: CCI(x.high, x.low, x.close, timeperiod=t)))
```

1.5.11 Chande Momentum Oscillator

[33]:
```
for t in T:
    universe[f'{t:02}_CMO'] = universe.groupby(level='symbol').close.apply(CMO,␣
↪timeperiod=t)
```

1.5.12 Rate of Change

Rate of change is a technical indicator that illustrates the speed of price change over a period of time.

[34]:
```
for t in T:
    universe[f'{t:02}_ROC'] = universe.groupby(level='symbol').close.apply(ROC,␣
↪timeperiod=t)
```

1.5.13 Chaikin A/D Oscillator

[35]:
```
for t in T:
    universe[f'{t:02}_ADOSC'] = (universe.groupby(level='symbol',␣
↪group_keys=False)
     .apply(lambda x: ADOSC(x.high, x.low, x.close, x.volume, fastperiod=t-3,␣
↪slowperiod=4+t)))
```

1.5.14 Average Directional Movement Index

[36]:
```
for t in T:
    universe[f'{t:02}_ADX'] = universe.groupby(level='symbol',
                                                group_keys=False).apply(lambda x:
                                                                       ADX(x.high, x.low, x.
↪close, timeperiod=t))
```

[37]:
```
universe.drop(ohlcv, axis=1).to_hdf('data.h5', 'features')
```

1.6 Compute Historical Returns

1.6.1 Historical Returns

[38]:
```
by_sym = universe.groupby(level='symbol').close
for t in [1,5]:
    universe[f'r{t:02}'] = by_sym.pct_change(t)
```

1.6.2 Remove outliers

[39]:
```
universe[[f'r{t:02}' for t in [1, 5]]].describe()
```

[39]:

	r01	r05
count	2.529493e+06	2.526553e+06
mean	6.710840e-04	3.293540e-03
std	2.875355e-02	6.344951e-02
min	-9.718670e-01	-9.795396e-01
25%	-1.034141e-02	-2.246575e-02
50%	3.236246e-04	2.921130e-03
75%	1.122661e-02	2.811951e-02
max	1.216425e+01	1.252657e+01

[40]:
```
outliers = universe[universe.r01>1].index.get_level_values('symbol').unique()
len(outliers)
```

[40]: 11

[41]:
```
universe = universe.drop(outliers, level='symbol')
```

1.6.3 Historical return quantiles

[42]:
```
for t in [1, 5]:
    universe[f'r{t:02}dec'] = (universe[f'r{t:02}'].groupby(level='date')
             .apply(lambda x: pd.qcut(x, q=10, labels=False,␣
 ↪duplicates='drop')))
```

1.7 Rolling Factor Betas

We also use five Fama-French risk factors (Fama and French, 2015; see Chapter 4, Financial Feature Engineering – How to Research Alpha Factors). They reflect the sensitivity of a stock's returns to factors consistently demonstrated to impact equity returns.

We capture these factors by computing the coefficients of a rolling OLS regression of a stock's daily returns on the returns of portfolios designed to reflect the underlying drivers: - **Equity risk premium**: Value-weighted returns of US stocks minus the 1-month US - **Treasury bill rate** - **Size (SMB)**: Returns of stocks categorized as Small (by market cap) Minus those of Big equities - **Value (HML)**: Returns of stocks with High book-to-market value Minus those with a Low value - **Investment (CMA)**: Returns differences for companies with Conservative investment expenditures Minus those with Aggressive spending - **Profitability (RMW)**: Similarly, return differences for stocks with Robust profitability Minus that with a Weak metric.

[43]:
```
factor_data = (web.DataReader('F-F_Research_Data_5_Factors_2x3_daily',␣
↪'famafrench',
                              start=START)[0].rename(columns={'Mkt-RF':␣
↪'Market'}))
factor_data.index.names = ['date']
```

[44]:
```
factor_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5284 entries, 2000-01-03 to 2020-12-31
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Market  5284 non-null   float64
 1   SMB     5284 non-null   float64
 2   HML     5284 non-null   float64
 3   RMW     5284 non-null   float64
 4   CMA     5284 non-null   float64
 5   RF      5284 non-null   float64
dtypes: float64(6)
memory usage: 289.0 KB
```

[45]:
```
windows = list(range(15, 90, 5))
len(windows)
```

[45]: 15

Next, we apply `statsmodels`' `RollingOLS()` to run regressions over windowed periods of different lengths, ranging from 15 to 90 days. We set the `params_only` parameter on the `.fit()` method to speed up computation and capture the coefficients using the `.params` attribute of the fitted factor_model:

[46]:
```
t = 1
ret = f'r{t:02}'
factors = ['Market', 'SMB', 'HML', 'RMW', 'CMA']
windows = list(range(15, 90, 5))
for window in windows:
    print(window)
    betas = []
    for symbol, data in universe.groupby(level='symbol'):
        model_data = data[[ret]].merge(factor_data, on='date').dropna()
        model_data[ret] -= model_data.RF

        rolling_ols = RollingOLS(endog=model_data[ret],
```

```
                                        exog=sm.add_constant(model_data[factors]),␣
↪window=window)
        factor_model = rolling_ols.fit(params_only=True).params.drop('const',␣
↪axis=1)
        result = factor_model.assign(symbol=symbol).set_index('symbol',␣
↪append=True)
        betas.append(result)
    betas = pd.concat(betas).rename(columns=lambda x: f'{window:02}_{x}')
    universe = universe.join(betas)
```

```
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
```

1.8 Compute Forward Returns

[47]:
```
for t in [1, 5]:
    universe[f'r{t:02}_fwd'] = universe.groupby(level='symbol')[f'r{t:02}'].
↪shift(-t)
    universe[f'r{t:02}dec_fwd'] = universe.groupby(level='symbol')[f'r{t:
↪02}dec'].shift(-t)
```

1.9 Store Model Data

[48]:
```
universe = universe.drop(ohlcv, axis=1)
```

[49]:
```
universe.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 2499265 entries, ('BRK_A', Timestamp('2000-01-03 00:00:00')) to
('BLL', Timestamp('2017-12-29 00:00:00'))
Columns: 308 entries, 06_RSI to r05dec_fwd
dtypes: float64(308)
memory usage: 5.7+ GB
```

[50]:
```
drop_cols = ['r01', 'r01dec', 'r05',  'r05dec']
```

[51]:
```
outcomes = universe.filter(like='_fwd').columns
```

[52]:
```
universe = universe.sort_index()
with pd.HDFStore('data.h5') as store:
    store.put('features', universe.drop(drop_cols, axis=1).drop(outcomes,␣
↪axis=1).loc[idx[:, '2001':], :])
    store.put('targets', universe.loc[idx[:, '2001':], outcomes])
```