PART IV
Algorithmic Trading

> Success means making profits and avoiding losses.
>
> —Martin Zweig

Part III is concerned with the discovery of *statistical* inefficiencies in financial markets by the use of deep learning and reinforcement learning techniques. This part, by contrast, is concerned with identifying and exploiting *economic* inefficiencies for which statistical inefficiencies are a prerequisite in general. The tool of choice for exploiting economic inefficiencies is *algorithmic trading*, that is, the automated execution of trading strategies based on predictions generated by a trading bot.

Table IV-1 compares in a simplified manner the problem of training and deploying a trading bot with the one of building and deploying a self-driving car.

Table IV-1. Self-driving cars compared to trading bots

Step	Self-Driving Car	Trading Bot
Training	Training AI in virtual and recorded environments	Training AI with simulated and real historical data
Risk management	Adding rules to avoid collisions, crashes, and so on	Adding rules to avoid large losses, to take profits early, and so on
Deployment	Combining AI with car hardware, deploying the car on the street, and monitoring	Combining AI with trading platform, deploying the trading bot for real trading, and monitoring

This part consists of three chapters that are structured along the three steps, as illustrated in Table IV-1, to exploit economic inefficiencies through a trading bot—starting with the vectorized backtesting of trading strategies, covering the analysis of risk

management measures through event-based backtesting, and discussing technical details in the context of strategy execution and deployment:

- Chapter 10 is about the *vectorized backtesting* of algorithmic trading strategies, such as those based on a DNN for market prediction. This approach is both efficient and insightful with regard to a first judgment of the economic potential of a trading strategy. It also allows one to assess the impact of transaction costs on economic performance.
- Chapter 11 covers central aspects of managing the risk of algorithmic trading strategies, such as the use of stop loss orders or take profit orders. In addition to vectorized backtesting, this chapter introduces event-based backtesting as a more flexible approach to judge the economic potential of a trading strategy.
- Chapter 12 is primarily about the execution of trading strategies. Topics are the retrieval of historical data, the training of a trading bot based on this data, the streaming of real-time data, and the placement of orders. It introduces Oanda and its API as a trading platform well suited to algorithmic trading. It also covers fundamental aspects of deploying AI-powered algorithmic trading strategies in automatic fashion.

Algorithmic Trading Strategies

Algorithmic trading is a vast field and encompasses different types of trading strategies. Some, for example, try to minimize the market impact during the execution of large orders (liquidity algorithms). Others try to replicate the payoff of derivatives instruments as closely as possible (dynamic hedging/replication). These examples illustrate that not all algorithmic trading strategies have the goal of exploiting economic inefficiencies. For the purposes of this book, the focus on algorithmic trading strategies that result from predictions made by a trading bot (for example, in the form of a DNN agent or an RL agent) seems appropriate and useful.

CHAPTER 10
Vectorized Backtesting

> Tesla's chief executive and serial technology entrepreneur, Elon Musk, has said his company's cars will be able to be summoned and drive autonomously across the US to pick up their owners within the next two years.
>
> —Samuel Gibbs (2016)

> Big money is made in the stock market by being on the right side of the major moves.
>
> —Martin Zweig

The term *vectorized backtesting* refers to a technical approach to backtesting algorithmic trading strategies, such as those based on a dense neural network (DNN) for market prediction. The books by Hilpisch (2018, ch. 15; 2020, ch. 4) cover vectorized backtesting based on a number of concrete examples. *Vectorized* in this context refers to a programming paradigm that relies heavily or even exclusively on vectorized code (that is, code without any looping on the Python level). Vectorization of code is good practice with such packages such as `Numpy` or `pandas` in general and has been used intensively in previous chapters as well. The benefits of vectorized code are more concise and easy-to-read code, as well as faster execution in many important scenarios. On the other hand, it might not be as flexible in backtesting trading strategies as, for example, event-based backtesting, which is introduced and used in Chapter 11.

Having a good AI-powered predictor available that beats a simple baseline predictor is important but is generally not enough to generate *alpha* (that is, above-market returns, possibly adjusted for risk). For example, it is also important for a prediction-based trading strategy to predict the large market movements correctly and not just the majority of the (potentially pretty small) market movements. Vectorized backtesting is an easy and fast way of figuring out the economic potential of a trading strategy.

Compared to autonomous vehicles (AVs), vectorized backtesting is like testing the AI of AVs in virtual environments just to see how it performs "in general" in a risk-free environment. However, for the AI of an AV it is not only important to perform well *on average*, but it is also of paramount importance to see how it masters critical or even extreme situations. Such an AI is supposed to cause "zero casualties" on average, not 0.1 or 0.5. For a financial AI, it is similarly—even if not equally—important to get the large market movements correct. Whereas this chapter focuses on the pure performance of financial AI agents (trading bots), Chapter 11 goes deeper into risk assessment and the backtesting of standard risk measures.

"Backtesting an SMA-Based Strategy" on page 282 introduces vectorized backtesting based on a simple example using simple moving averages as technical indicators and end-of-day (EOD) data. This allows for insightful visualizations and an easier understanding of the approach when getting started. "Backtesting a Daily DNN-Based Strategy" on page 289 trains a DNN based on EOD data and backtests the resulting prediction-based strategy for its economic performance. "Backtesting an Intraday DNN-Based Strategy" on page 295 then does the same with intraday data. In all examples, proportional transaction costs are included in the form of assumed bid-ask spreads.

Backtesting an SMA-Based Strategy

This section introduces vectorized backtesting based on a classical trading strategy that uses simple moving averages (SMAs) as technical indicators. The following code realizes the necessary imports and configurations and retrieves EOD data for the EUR/USD currency pair:

```
In [1]: import os
        import math
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('mode.chained_assignment', None)
        pd.set_option('display.float_format', '{:.4f}'.format)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'

In [2]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'  ❶

In [3]: symbol = 'EUR='  ❶

In [4]: data = pd.DataFrame(pd.read_csv(url, index_col=0,
                                        parse_dates=True).dropna()[symbol])  ❶
```

```
In [5]: data.info()  ❶
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2516 entries, 2010-01-04 to 2019-12-31
        Data columns (total 1 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   EUR=    2516 non-null   float64
        dtypes: float64(1)
        memory usage: 39.3 KB
```

❶ Retrieves EOD data for EUR/USD

The idea of the strategy is the following. Calculate a shorter `SMA1`, say for 42 days, and a longer `SMA2`, say for 258 days. Whenever `SMA1` is above `SMA2`, go long on the financial instrument. Whenever `SMA1` is below `SMA2`, go short on the financial instrument. Because the example is based on EUR/USD, going long or short is easily accomplished.

The following Python code calculates in vectorized fashion the SMA values and visualizes the resulting time series alongside the original time series (see Figure 10-1):

```
In [6]: data['SMA1'] = data[symbol].rolling(42).mean()  ❶

In [7]: data['SMA2'] = data[symbol].rolling(258).mean()  ❷

In [8]: data.plot(figsize=(10, 6));  ❸
```

❶ Calculates the shorter `SMA1`

❷ Calculates the longer `SMA2`

❸ Visualizes the three time series

Equipped with the SMA time series data, the resulting positions can, again in vectorized fashion, be derived. Note the shift of the resulting position time series by one day to avoid foresight bias in the data. This shift is necessary since the calculation of the SMAs includes the closing values from the same day. Therefore, the position derived from the SMA values from one day needs to be applied to the next day for the whole time series.

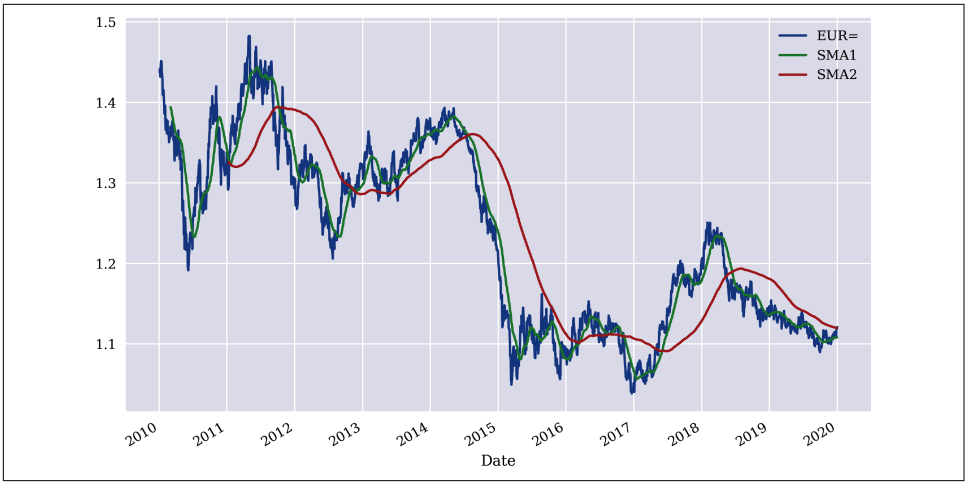

Figure 10-1. Time series data for EUR/USD and SMAs

Figure 10-2 visualizes the resulting positions as an overlay to the other time series:

```
In [9]: data.dropna(inplace=True)  ❶

In [10]: data['p'] = np.where(data['SMA1'] > data['SMA2'], 1, -1)  ❷

In [11]: data['p'] = data['p'].shift(1)  ❸

In [12]: data.dropna(inplace=True)  ❶

In [13]: data.plot(figsize=(10, 6), secondary_y='p');  ❹
```

❶ Deletes rows containing `NaN` values

❷ Derives the position values based on same-day SMA values

❸ Shifts the position values by one day to avoid foresight bias

❹ Visualizes the position values as derived from the SMAs

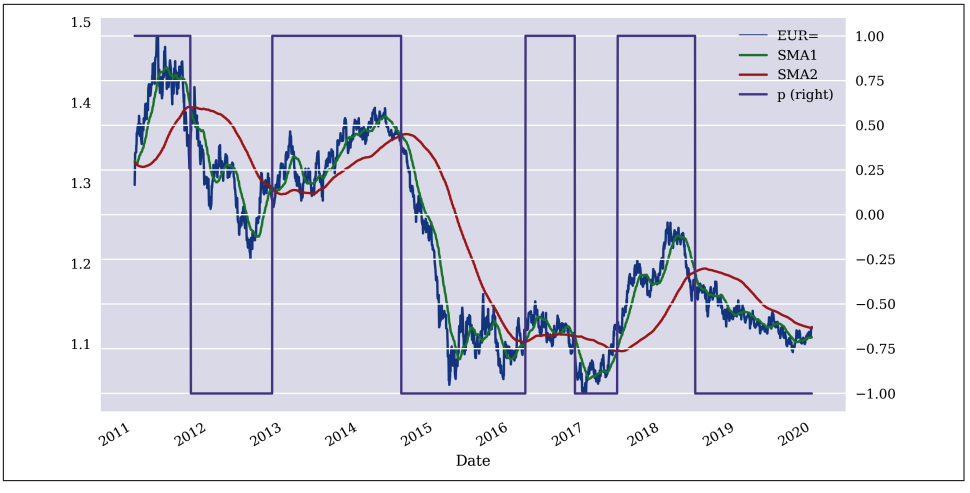

Figure 10-2. Time series data for EUR/USD, SMAs, and resulting positions

One crucial step is missing: the combination of the positions with the returns of the financial instrument. Since positions are conveniently represented by a `+1` for a long position and a `-1` for a short position, this step boils down to multiplying two columns of the `DataFrame` object—in vectorized fashion again. The SMA-based trading strategy outperforms the passive benchmark investment by a considerable margin, as Figure 10-3 illustrates:

```
In [14]: data['r'] = np.log(data[symbol] / data[symbol].shift(1))  ❶

In [15]: data.dropna(inplace=True)

In [16]: data['s'] = data['p'] * data['r']  ❷

In [17]: data[['r', 's']].sum().apply(np.exp)  ❸
Out[17]: r   0.8640
         s   1.3773
         dtype: float64

In [18]: data[['r', 's']].sum().apply(np.exp) - 1  ❹
Out[18]: r   -0.1360
         s    0.3773
         dtype: float64

In [19]: data[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));  ❺
```

❶ Calculates the log returns

❷ Calculates the strategy returns

❸ Calculates the gross performances

❹ Calculates the net performances

❺ Visualizes the gross performances over time

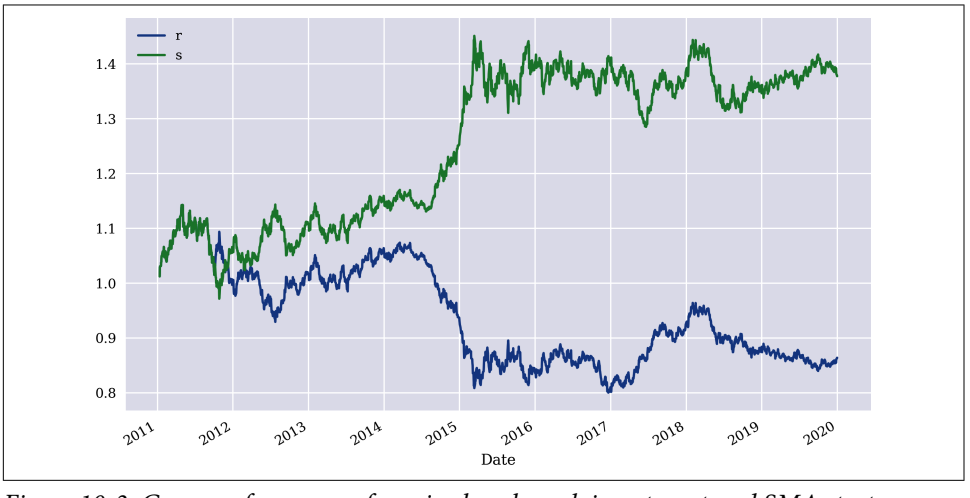

Figure 10-3. Gross performance of passive benchmark investment and SMA strategy

So far, the performance figures are not considering transaction costs. These are, of course, a crucial element when judging the economic potential of a trading strategy. In the current setup, proportional transaction costs can be easily included in the calculations. The idea is to determine when a trade takes place and to reduce the performance of the trading strategy by a certain value to account for the relevant bid-ask spread. As the following calculations show, and as is obvious from Figure 10-2, the trading strategy does not change positions too often. Therefore, in order to have some meaningful effects of transaction costs, they are assumed to be quite a bit higher than typically seen for EUR/USD. The net effect of subtracting transaction costs is a few percentage points under the given assumptions (see Figure 10-4):

```
In [20]: sum(data['p'].diff() != 0) + 2  ❶
Out[20]: 10

In [21]: pc = 0.005  ❷

In [22]: data['s_'] = np.where(data['p'].diff() != 0,
                               data['s'] - pc, data['s'])  ❸

In [23]: data['s_'].iloc[0] -= pc  ❹

In [24]: data['s_'].iloc[-1] -= pc  ❺

In [25]: data[['r', 's', 's_']][data['p'].diff() != 0]  ❻
Out[25]:                  r       s      s_
```

```
            Date
            2011-01-12  0.0123  0.0123  0.0023
            2011-10-10  0.0198 -0.0198 -0.0248
            2012-11-07 -0.0034 -0.0034 -0.0084
            2014-07-24 -0.0001  0.0001 -0.0049
            2016-03-16  0.0102  0.0102  0.0052
            2016-11-10 -0.0018  0.0018 -0.0032
            2017-06-05 -0.0025 -0.0025 -0.0075
            2018-06-15  0.0035 -0.0035 -0.0085

In [26]: data[['r', 's', 's_']].sum().apply(np.exp)
Out[26]: r    0.8640
         s    1.3773
         s_   1.3102
         dtype: float64

In [27]: data[['r', 's', 's_']].sum().apply(np.exp) - 1
Out[27]: r    -0.1360
         s     0.3773
         s_    0.3102
         dtype: float64

In [28]: data[['r', 's', 's_']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Calculates the number of trades, including entry and exit trade

❷ Fixes the proportional transaction costs (deliberately set quite high)

❸ Adjusts the strategy performance for the transaction costs

❹ Adjusts the strategy performance for the *entry* trade

❺ Adjusts the strategy performance for the *exit* trade

❻ Shows the adjusted performance values for the regular trades

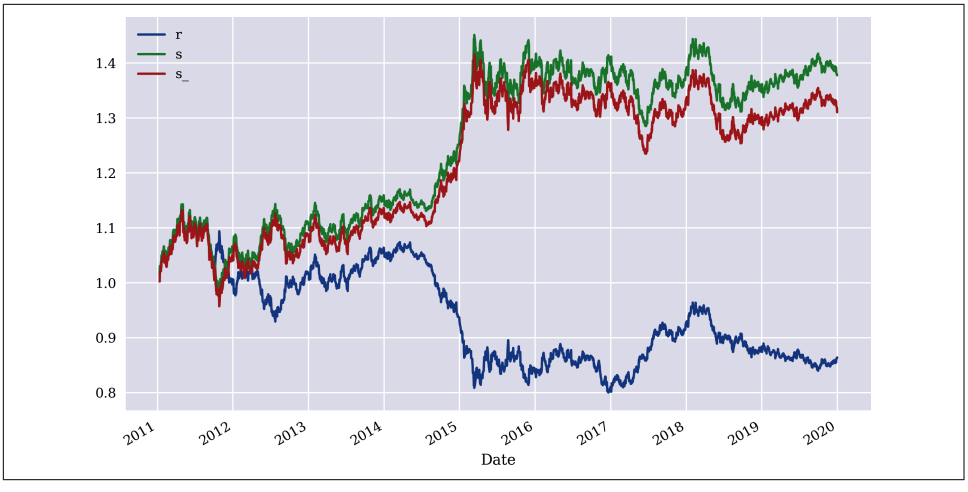

Figure 10-4. Gross performance of the SMA strategy before and after transaction costs

What about the resulting risk of the trading strategy? For a trading strategy that is based on directional predictions and that takes long or short positions only, the risk, expressed as the volatility (standard deviation of the log returns), is exactly the same as for the passive benchmark investment:

```
In [29]: data[['r', 's', 's_']].std()  ❶
Out[29]: r    0.0054
         s    0.0054
         s_   0.0054
         dtype: float64

In [30]: data[['r', 's', 's_']].std() * math.sqrt(252)  ❷
Out[30]: r    0.0853
         s    0.0853
         s_   0.0855
         dtype: float64
```

❶ Daily volatility

❷ Annualized volatility

Vectorized Backtesting

Vectorized backtesting is a powerful and efficient approach to backtesting the "pure" performance of a prediction-based trading strategy. It can also accommodate proportional transaction costs, for instance. However, it is not well suited to including typical risk management measures, such as (trailing) stop loss orders or take profit orders. This is addressed in Chapter 11.

Backtesting a Daily DNN-Based Strategy

The previous section lays out the blueprint for vectorized backtesting on the basis of a simple, easy-to-visualize trading strategy. The same blueprint can be applied, for example, to DNN-based trading strategies with minimal technical adjustments. The following trains a `Keras` DNN model, as discussed in Chapter 7. The data that is used is the same as in the previous example. However, as in Chapter 7, different features and lags thereof need to be added to the `DataFrame` object:

```
In [31]: data = pd.DataFrame(pd.read_csv(url, index_col=0,
                                         parse_dates=True).dropna()[symbol])

In [32]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2516 entries, 2010-01-04 to 2019-12-31
         Data columns (total 1 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   EUR=    2516 non-null   float64
         dtypes: float64(1)
         memory usage: 39.3 KB

In [33]: lags = 5

In [34]: def add_lags(data, symbol, lags, window=20):
             cols = []
             df = data.copy()
             df.dropna(inplace=True)
             df['r'] = np.log(df / df.shift(1))
             df['sma'] = df[symbol].rolling(window).mean()
             df['min'] = df[symbol].rolling(window).min()
             df['max'] = df[symbol].rolling(window).max()
             df['mom'] = df['r'].rolling(window).mean()
             df['vol'] = df['r'].rolling(window).std()
             df.dropna(inplace=True)
             df['d'] = np.where(df['r'] > 0, 1, 0)
             features = [symbol, 'r', 'd', 'sma', 'min', 'max', 'mom', 'vol']
             for f in features:
                 for lag in range(1, lags + 1):
                     col = f'{f}_lag_{lag}'
                     df[col] = df[f].shift(lag)
                     cols.append(col)
             df.dropna(inplace=True)
             return df, cols

In [35]: data, cols = add_lags(data, symbol, lags, window=20)
```

The following Python code accomplishes additional imports and defines the `set_seeds()` and `create_model()` functions:

```
In [36]: import random
         import tensorflow as tf
         from keras.layers import Dense, Dropout
         from keras.models import Sequential
         from keras.regularizers import l1
         from keras.optimizers import Adam
         from sklearn.metrics import accuracy_score
         Using TensorFlow backend.

In [37]: def set_seeds(seed=100):
             random.seed(seed)
             np.random.seed(seed)
             tf.random.set_seed(seed)
         set_seeds()

In [38]: optimizer = Adam(learning_rate=0.0001)

In [39]: def create_model(hl=2, hu=128, dropout=False, rate=0.3,
                          regularize=False, reg=l1(0.0005),
                          optimizer=optimizer, input_dim=len(cols)):
             if not regularize:
                 reg = None
             model = Sequential()
             model.add(Dense(hu, input_dim=input_dim,
                         activity_regularizer=reg,
                         activation='relu'))
             if dropout:
                 model.add(Dropout(rate, seed=100))
             for _ in range(hl):
                 model.add(Dense(hu, activation='relu',
                                 activity_regularizer=reg))
                 if dropout:
                     model.add(Dropout(rate, seed=100))
             model.add(Dense(1, activation='sigmoid'))
             model.compile(loss='binary_crossentropy',
                           optimizer=optimizer,
                           metrics=['accuracy'])
             return model
```

Based on a sequential train-test split of the historical data, the following Python code first trains the DNN model based on normalized features data:

```
In [40]: split = '2018-01-01'  ❶

In [41]: train = data.loc[:split].copy()  ❶

In [42]: np.bincount(train['d'])  ❷
Out[42]: array([ 982, 1006])
```

```
In [43]: mu, std = train.mean(), train.std()  ❸

In [44]: train_ = (train - mu) / std  ❸

In [45]: set_seeds()
         model = create_model(hl=2, hu=64)  ❹

In [46]: %%time
         model.fit(train_[cols], train['d'],
                 epochs=20, verbose=False,
                 validation_split=0.2, shuffle=False)  ❺
         CPU times: user 2.93 s, sys: 574 ms, total: 3.5 s
         Wall time: 1.93 s

Out[46]: <keras.callbacks.callbacks.History at 0x7fc9392f38d0>

In [47]: model.evaluate(train_[cols], train['d'])  ❻
         1988/1988 [==============================] - 0s 17us/step

Out[47]: [0.6745863538872549, 0.5925553441047668]
```

❶ Splits the data into training and test data

❷ Shows the frequency of the labels classes

❸ Normalizes the training features data

❹ Creates the DNN model

❺ Trains the DNN model on the training data

❻ Evaluates the performance of the model on the training data

So far, this basically repeats the core approach of Chapter 7. Vectorized backtesting can now be applied to judge the economic performance of the DNN-based trading strategy *in-sample* based on the model's predictions (see Figure 10-5). In this context, an upward prediction is naturally interpreted as a long position and a downward prediction as a short position:

```
In [48]: train['p'] = np.where(model.predict(train_[cols]) > 0.5, 1, 0)  ❶

In [49]: train['p'] = np.where(train['p'] == 1, 1, -1)  ❷

In [50]: train['p'].value_counts()  ❸
Out[50]: -1    1098
          1     890
         Name: p, dtype: int64

In [51]: train['s'] = train['p'] * train['r']  ❹
```

```
In [52]: train[['r', 's']].sum().apply(np.exp)  ❺
Out[52]: r   0.8787
         s   5.0766
         dtype: float64

In [53]: train[['r', 's']].sum().apply(np.exp)  - 1  ❺
Out[53]: r   -0.1213
         s    4.0766
         dtype: float64

In [54]: train[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));  ❻
```

❶ Generates the binary predictions

❷ Translates the predictions into position values

❸ Shows the number of long and short positions

❹ Calculates the strategy performance values

❺ Calculates the gross and net performances (in-sample)

❻ Visualizes the gross performances over time (in-sample)

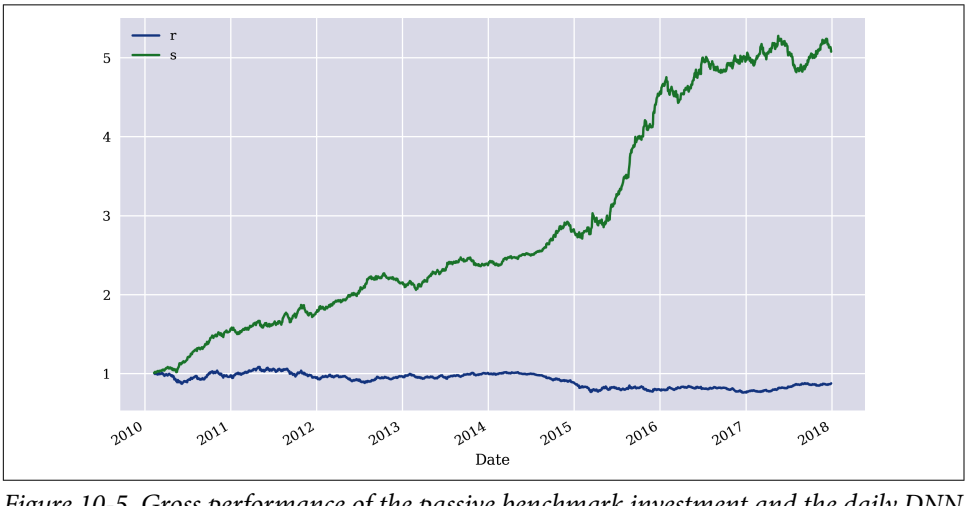

Figure 10-5. Gross performance of the passive benchmark investment and the daily DNN strategy (in-sample)

Next is the same sequence of calculations for the test data set. Whereas the out-performance in-sample is significant, the numbers out-of-sample are not as impressive but are still convincing (see Figure 10-6):

```
In [55]: test = data.loc[split:].copy()  ❶

In [56]: test_ = (test - mu) / std  ❷

In [57]: model.evaluate(test_[cols], test['d'])  ❸
         503/503 [==============================] - 0s 17us/step

Out[57]: [0.6933823573897421, 0.5407554507255554]

In [58]: test['p'] = np.where(model.predict(test_[cols]) > 0.5, 1, -1)

In [59]: test['p'].value_counts()
Out[59]: -1    406
          1     97
         Name: p, dtype: int64

In [60]: test['s'] = test['p'] * test['r']

In [61]: test[['r', 's']].sum().apply(np.exp)
Out[61]: r   0.9345
         s   1.2431
         dtype: float64

In [62]: test[['r', 's']].sum().apply(np.exp) - 1
Out[62]: r   -0.0655
         s    0.2431
         dtype: float64

In [63]: test[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Generates the test data sub-set

❷ Normalizes the test data

❸ Evaluates the model performance on the test data

The DNN-based trading strategy leads to a larger number of trades as compared to the SMA-based strategy. This makes the inclusion of transaction costs an even more important aspect when judging the economic performance.

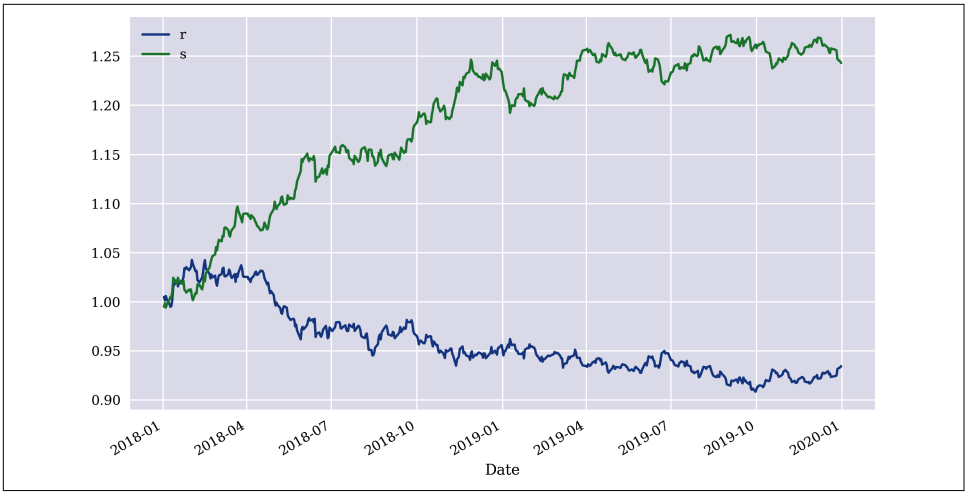

Figure 10-6. Gross performance of the passive benchmark investment and the daily DNN strategy (out-of-sample)

The following code assumes now realistic bid-ask spreads for EUR/USD on the level of 1.2 pips (that is, 0.00012 in terms of currency units).[1] To simplify the calculations, an average value for the proportional transaction costs `pc` is calculated based on the average closing price for EUR/USD (see Figure 10-7):

```
In [64]: sum(test['p'].diff() != 0)
Out[64]: 147

In [65]: spread = 0.00012  ❶
         pc = spread / data[symbol].mean()  ❷
         print(f'{pc:.6f}')
         0.000098

In [66]: test['s_'] = np.where(test['p'].diff() != 0,
                               test['s'] - pc, test['s'])

In [67]: test['s_'].iloc[0] -= pc

In [68]: test['s_'].iloc[-1] -= pc

In [69]: test[['r', 's', 's_']].sum().apply(np.exp)
Out[69]: r    0.9345
         s    1.2431
         s_   1.2252
         dtype: float64
```

1 This, for example, is a typical spread that Oanda offers retail traders.

```
In [70]: test[['r', 's', 's_']].sum().apply(np.exp) - 1
Out[70]: r    -0.0655
         s     0.2431
         s_    0.2252
         dtype: float64

In [71]: test[['r', 's', 's_']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Fixes the average bid-ask spread

❷ Calculates the average proportional transaction costs

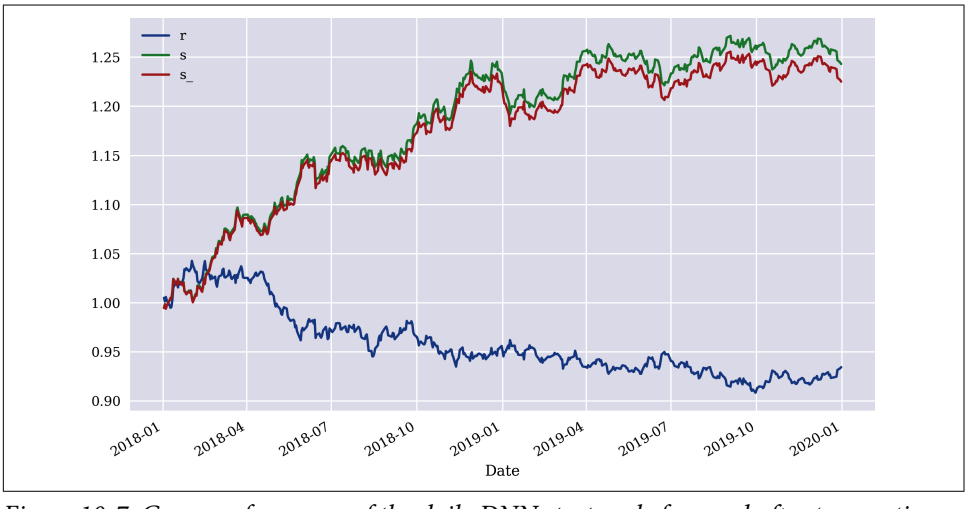

Figure 10-7. Gross performance of the daily DNN strategy before and after transaction costs (out-of-sample)

The DNN-based trading strategy seems promising both before and after typical transaction costs. However, would a similar strategy be economically viable intraday as well, when even more trades are observed? The next section analyzes a DNN-based intraday strategy.

Backtesting an Intraday DNN-Based Strategy

To train and backtest a DNN model on intraday data, another data set is required:

```
In [72]: url = 'http://hilpisch.com/aiif_eikon_id_eur_usd.csv'  ❶

In [73]: symbol = 'EUR='  ❶

In [74]: data = pd.DataFrame(pd.read_csv(url, index_col=0,
                             parse_dates=True).dropna()['CLOSE'])  ❶
         data.columns = [symbol]
```

```
In [75]: data = data.resample('5min', label='right').last().ffill()  ❷

In [76]: data.info()  ❷
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 26486 entries, 2019-10-01 00:05:00 to 2019-12-31 23:10:00
         Freq: 5T
         Data columns (total 1 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   EUR=    26486 non-null  float64
         dtypes: float64(1)
         memory usage: 413.8 KB

In [77]: lags = 5

In [78]: data, cols = add_lags(data, symbol, lags, window=20)
```

❶ Retrieves intraday data for EUR/USD and picks the closing prices

❷ Resamples the data to five-minute bars

The procedure of the previous section can now be repeated with the new data set. First, train the DNN model:

```
In [79]: split = int(len(data) * 0.85)

In [80]: train = data.iloc[:split].copy()

In [81]: np.bincount(train['d'])
Out[81]: array([16284,  6207])

In [82]: def cw(df):
             c0, c1 = np.bincount(df['d'])
             w0 = (1 / c0) * (len(df)) / 2
             w1 = (1 / c1) * (len(df)) / 2
             return {0: w0, 1: w1}

In [83]: mu, std = train.mean(), train.std()

In [84]: train_ = (train - mu) / std

In [85]: set_seeds()
         model = create_model(hl=1, hu=128,
                              reg=True, dropout=False)

In [86]: %%time
         model.fit(train_[cols], train['d'],
                   epochs=40, verbose=False,
                   validation_split=0.2, shuffle=False,
                   class_weight=cw(train))
         CPU times: user 40.6 s, sys: 5.49 s, total: 46 s
         Wall time: 25.2 s
```

```
Out[86]: <keras.callbacks.callbacks.History at 0x7fc91a6b2a90>

In [87]: model.evaluate(train_[cols], train['d'])
         22491/22491 [==============================] - 0s 13us/step

Out[87]: [0.5218664327576152, 0.6729803085327148]
```

In-sample, the performance looks promising, as illustrated in Figure 10-8:

```
In [88]: train['p'] = np.where(model.predict(train_[cols]) > 0.5, 1, -1)

In [89]: train['p'].value_counts()
Out[89]: -1    11519
          1    10972
         Name: p, dtype: int64

In [90]: train['s'] = train['p'] * train['r']

In [91]: train[['r', 's']].sum().apply(np.exp)
Out[91]: r   1.0223
         s   1.6665
         dtype: float64

In [92]: train[['r', 's']].sum().apply(np.exp) - 1
Out[92]: r   0.0223
         s   0.6665
         dtype: float64

In [93]: train[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

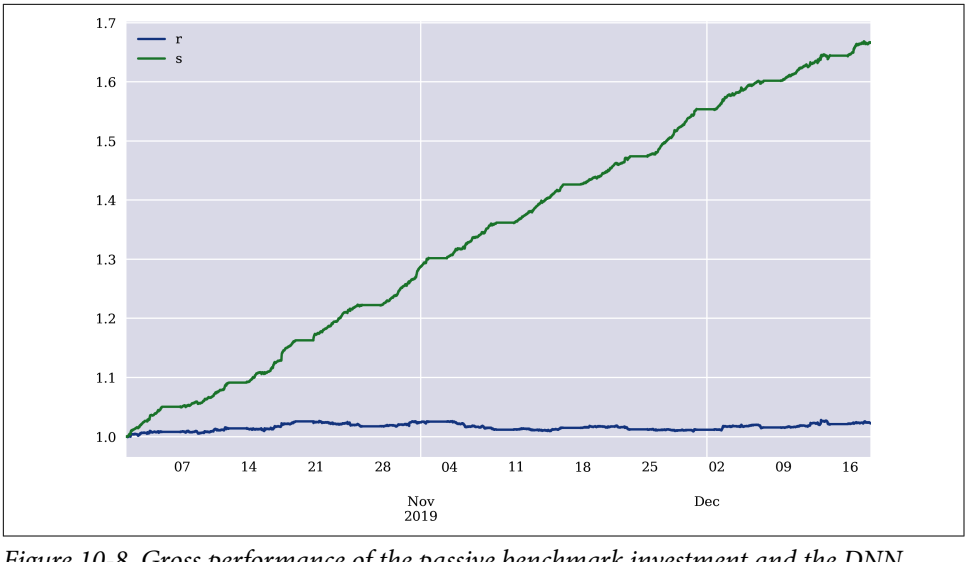

Figure 10-8. Gross performance of the passive benchmark investment and the DNN intraday strategy (in-sample)

Out-of-sample, the performance also looks promising before transaction costs. The strategy seems to systematically outperform the passive benchmark investment (see Figure 10-9):

```
In [94]: test = data.iloc[split:].copy()

In [95]: test_ = (test - mu) / std

In [96]: model.evaluate(test_[cols], test['d'])
         3970/3970 [==============================] - 0s 19us/step

Out[96]: [0.5226116042706168, 0.668513834476471]

In [97]: test['p'] = np.where(model.predict(test_[cols]) > 0.5, 1, -1)

In [98]: test['p'].value_counts()
Out[98]: -1    2273
          1    1697
         Name: p, dtype: int64

In [99]: test['s'] = test['p'] * test['r']

In [100]: test[['r', 's']].sum().apply(np.exp)
Out[100]: r   1.0071
          s   1.0658
          dtype: float64

In [101]: test[['r', 's']].sum().apply(np.exp) - 1
Out[101]: r   0.0071
          s   0.0658
          dtype: float64

In [102]: test[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

The final litmus test with regard to pure economic performance comes when adding transaction costs. The strategy leads to hundreds of trades over a relatively short period of time. As the following analysis suggests, based on standard retail bid-ask spreads, the DNN-based strategy is not viable.

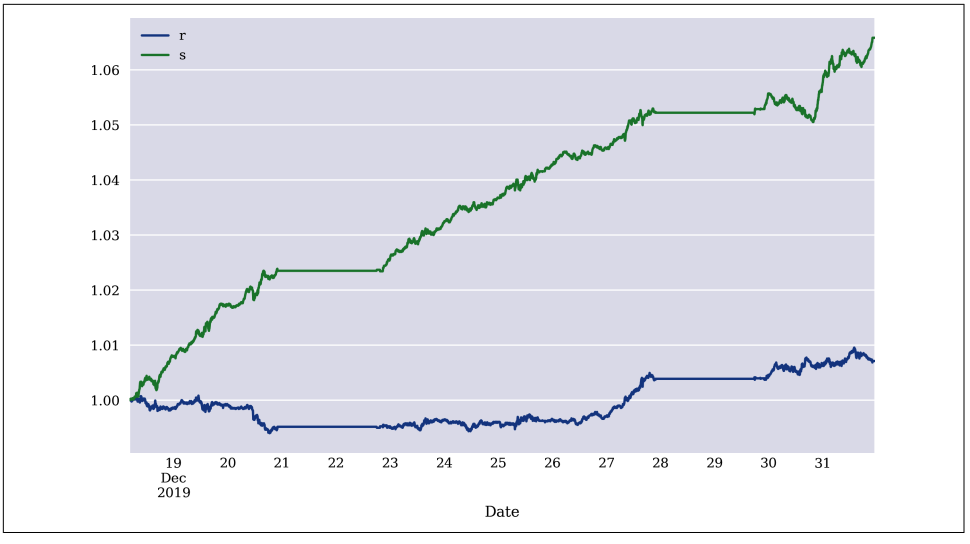

Figure 10-9. Gross performance of the passive benchmark investment and the DNN intraday strategy (out-of-sample)

Reducing the spread to a level that professional, high-volume traders might achieve, the strategy still does not break even but rather loses a large proportion of the profits to the transaction costs (see Figure 10-10):

```
In [103]: sum(test['p'].diff() != 0)
Out[103]: 1303

In [104]: spread = 0.00012  ❶
          pc_1 = spread / test[symbol]  ❶

In [105]: spread = 0.00006  ❷
          pc_2 = spread / test[symbol]  ❷

In [106]: test['s_1'] = np.where(test['p'].diff() != 0,
                                 test['s'] - pc_1, test['s'])  ❶

In [107]: test['s_1'].iloc[0] -= pc_1.iloc[0]  ❶
          test['s_1'].iloc[-1] -= pc_1.iloc[0]  ❶

In [108]: test['s_2'] = np.where(test['p'].diff() != 0,
                                 test['s'] - pc_2, test['s'])  ❷

In [109]: test['s_2'].iloc[0] -= pc_2.iloc[0]  ❷
          test['s_2'].iloc[-1] -= pc_2.iloc[0]  ❷

In [110]: test[['r', 's', 's_1', 's_2']].sum().apply(np.exp)
Out[110]: r     1.0071
          s     1.0658
```

```
s_1   0.9259
s_2   0.9934
dtype: float64
```

```
In [111]: test[['r', 's', 's_1', 's_2']].sum().apply(np.exp) - 1
Out[111]: r      0.0071
          s      0.0658
          s_1   -0.0741
          s_2   -0.0066
          dtype: float64
```

```
In [112]: test[['r', 's', 's_1', 's_2']].cumsum().apply(
              np.exp).plot(figsize=(10, 6), style=['-', '-', '--', '--']);
```

❶ Assumes bid-ask spread on retail level

❷ Assumes bid-ask spread on professional level

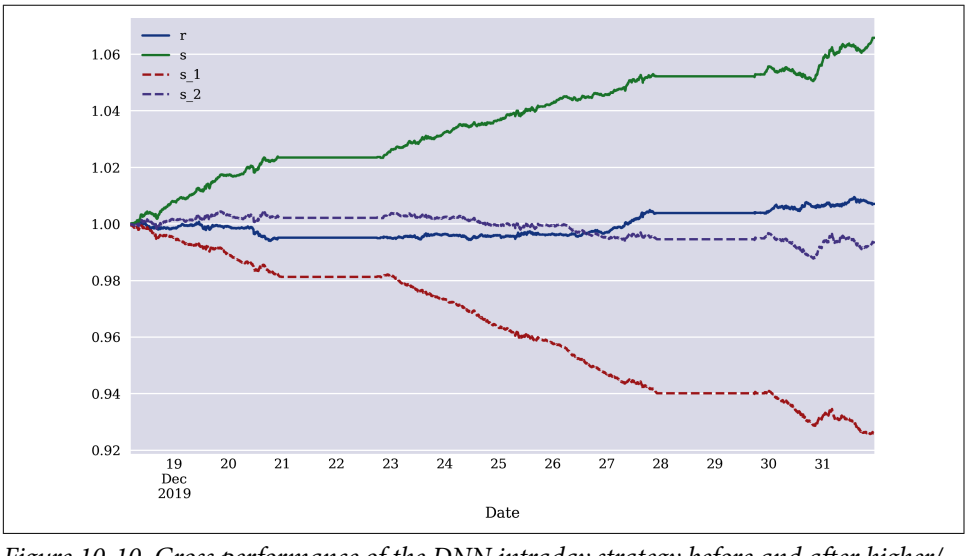

Figure 10-10. Gross performance of the DNN intraday strategy before and after higher/lower transaction costs (out-of-sample)

Intraday Trading

Intraday algorithmic trading in the form discussed in this chapter often seems appealing from a statistical point of view. Both in-sample and out-of-sample, the DNN model reaches a high accuracy when predicting the market direction. Excluding transaction costs, this also translates both in-sample and out-of-sample into a significant outperformance of the DNN-based strategy when compared to the passive benchmark investment. However, adding transaction costs to the mix reduces the performance of the DNN-based strategy considerably, making it unviable for typical retail bid-ask spreads and not really attractive for lower, high-volume bid-ask spreads.

Conclusions

Vectorized backtesting proves to be an efficient and valuable approach for backtesting the performance of AI-powered algorithmic trading strategies. This chapter first explains the basic idea behind the approach based on a simple example using two SMAs to derive signals. This allows for a simple visualization of the strategy and resulting positions. It then proceeds by backtesting a DNN-based trading strategy, as discussed in detail in Chapter 7, in combination with EOD data. Both before and after transaction costs, the *statistical inefficiencies* as discovered in Chapter 7 translate into *economic inefficiencies*, which means profitable trading strategies. When using the same vectorized backtesting approaches with intraday data, the DNN strategy also shows a significant outperformance both in- and out-of-sample when compared to the passive benchmark investment—at least before transaction costs. Adding transaction costs to the backtesting illustrates that these must be pretty low, on a level often not even achieved by big professional traders, to render the trading strategy economically viable.

References

Books and papers cited in this chapter:

Gibbs Samuel. 2016. "Elon Musk: Tesla Cars Will Be Able to Cross Us with No Driver in Two Years." *The Guardian*. January 11, 2016. *https://oreil.ly/C508Q*.

Hilpisch, Yves. 2018. *Python for Finance: Mastering Data-Driven Finance.* 2nd ed. Sebastopol: O'Reilly.

———. 2020. *Python for Algorithmic Trading: From Idea to Cloud Deployment.* Sebastopol: O'Reilly.

CHAPTER 11
Risk Management

> A significant barrier to deploying autonomous vehicles (AVs) on a massive scale is safety assurance.
>
> —Majid Khonji et al. (2019)

> Having better prediction raises the value of judgment. After all, it doesn't help to know the likelihood of rain if you don't know how much you like staying dry or how much you hate carrying an umbrella.
>
> —Ajay Agrawal et al. (2018)

Vectorized backtesting in general enables one to judge the economic potential of a prediction-based algorithmic trading strategy on an as-is basis (that is, in its pure form). Most AI agents applied in practice have more components than just the prediction model. For example, the AI of autonomous vehicles (AVs) comes not standalone but rather with a large number of rules and heuristics that restrict what actions the AI takes or can take. In the context of AVs, this primarily relates to managing risks, such as those resulting from collisions or crashes.

In a financial context, AI agents or trading bots are also not deployed as-is in general. Rather, there are a number of standard risk measures that are typically used, such as *(trailing) stop loss orders* or *take profit orders*. The reasoning is clear. When placing directional bets in financial markets, too-large losses are to be avoided. Similarly, when a certain profit level is reached, the success is to be protected by early close outs. How such risk measures are handled is a matter, more often than not, of human judgment, supported probably by a formal analysis of relevant data and statistics. Conceptually, this is a major point discussed in the book by Agrawal et al. (2018): AI provides improved predictions, but human judgment still plays a role in setting decision rules and action boundaries.

This chapter has a threefold purpose. First, it backtests in both *vectorized* and *event-based* fashion algorithmic trading strategies that result from a trained deep Q-learning agent. Henceforth, such agents are called *trading bots*. Second, it assesses risks related to the financial instrument on which the strategies are implemented. And third, it backtests typical risk measures, such as stop loss orders, using the event-based approach introduced in this chapter. The major benefit of event-based backtesting when compared to vectorized backtesting is a higher degree of flexibility in modeling and analyzing decision rules and risk management measures. In other words, it allows one to zoom in on details that are pushed toward the background when working with vectorized programming approaches.

"Trading Bot" on page 304 introduces and trains the trading bot based on the financial Q-learning agent from Chapter 9. "Vectorized Backtesting" on page 308 uses vectorized backtesting from Chapter 10 to judge the (pure) economic performance of the trading bot. Event-based backtesting is introduced in "Event-Based Backtesting" on page 311. First, a base class is discussed. Second, based on the base class, the backtesting of the trading bot is implemented and conducted. In this context, also see Hilpisch (2020, ch. 6). "Assessing Risk" on page 318 analyzes selected statistical measures important for setting risk management rules, such as *maximum drawdown* and *average true range* (ATR). "Backtesting Risk Measures" on page 322 then backtests the impact of major risk measures on the performance of the trading bot.

Trading Bot

This section presents a trading bot based on the financial Q-learning agent, `FQLAgent`, from Chapter 9. This is the trading bot that is analyzed in subsequent sections. As usual, our imports come first:

```
In [1]: import os
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('mode.chained_assignment', None)
        pd.set_option('display.float_format', '{:.4f}'.format)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'
```

"Finance Environment" on page 333 presents a Python module with the `Finance` class used in the following. "Trading Bot" on page 304 provides the Python module with the `TradingBot` class and some helper functions for plotting training and validation results. Both classes are pretty close to the ones introduced in Chapter 9, which is why they are used here without further explanations.

The following code trains the trading bot on historical end-of-day (EOD) data, including a sub-set of the data used for validation. Figure 11-1 shows average total rewards as achieved for the different training episodes:

```
In [2]: import finance
        import tradingbot
        Using TensorFlow backend.

In [3]: symbol = 'EUR='
        features = [symbol, 'r', 's', 'm', 'v']

In [4]: a = 0
        b = 1750
        c = 250

In [5]: learn_env = finance.Finance(symbol, features, window=20, lags=3,
                            leverage=1, min_performance=0.9, min_accuracy=0.475,
                            start=a, end=a + b, mu=None, std=None)

In [6]: learn_env.data.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 1750 entries, 2010-02-02 to 2017-01-12
        Data columns (total 6 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   EUR=    1750 non-null   float64
         1   r       1750 non-null   float64
         2   s       1750 non-null   float64
         3   m       1750 non-null   float64
         4   v       1750 non-null   float64
         5   d       1750 non-null   int64
        dtypes: float64(5), int64(1)
        memory usage: 95.7 KB

In [7]: valid_env = finance.Finance(symbol, features=learn_env.features,
                                    window=learn_env.window,
                                    lags=learn_env.lags,
                                    leverage=learn_env.leverage,
                                    min_performance=0.0, min_accuracy=0.0,
                                    start=a + b, end=a + b + c,
                                    mu=learn_env.mu, std=learn_env.std)

In [8]: valid_env.data.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 250 entries, 2017-01-13 to 2018-01-10
        Data columns (total 6 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   EUR=    250 non-null    float64
         1   r       250 non-null    float64
         2   s       250 non-null    float64
         3   m       250 non-null    float64
```

```
 4   v       250 non-null    float64
 5   d       250 non-null    int64
dtypes: float64(5), int64(1)
memory usage: 13.7 KB

In [9]: tradingbot.set_seeds(100)
        agent = tradingbot.TradingBot(24, 0.001, learn_env, valid_env)

In [10]: episodes = 61

In [11]: %time agent.learn(episodes)
         ============================================================================
         episode: 10/61 | VALIDATION | treward:  247 | perf: 0.936 | eps: 0.95
         ============================================================================
         ============================================================================
         episode: 20/61 | VALIDATION | treward:  247 | perf: 0.897 | eps: 0.86
         ============================================================================
         ============================================================================
         episode: 30/61 | VALIDATION | treward:  247 | perf: 1.035 | eps: 0.78
         ============================================================================
         ============================================================================
         episode: 40/61 | VALIDATION | treward:  247 | perf: 0.935 | eps: 0.70
         ============================================================================
         ============================================================================
         episode: 50/61 | VALIDATION | treward:  247 | perf: 0.890 | eps: 0.64
         ============================================================================
         ============================================================================
         episode: 60/61 | VALIDATION | treward:  247 | perf: 0.998 | eps: 0.58
         ============================================================================
         episode: 61/61 | treward:   17 | perf: 0.979 | av: 475.1 | max: 1747
         CPU times: user 51.4 s, sys: 2.53 s, total: 53.9 s
         Wall time: 47 s

In [12]: tradingbot.plot_treward(agent)
```

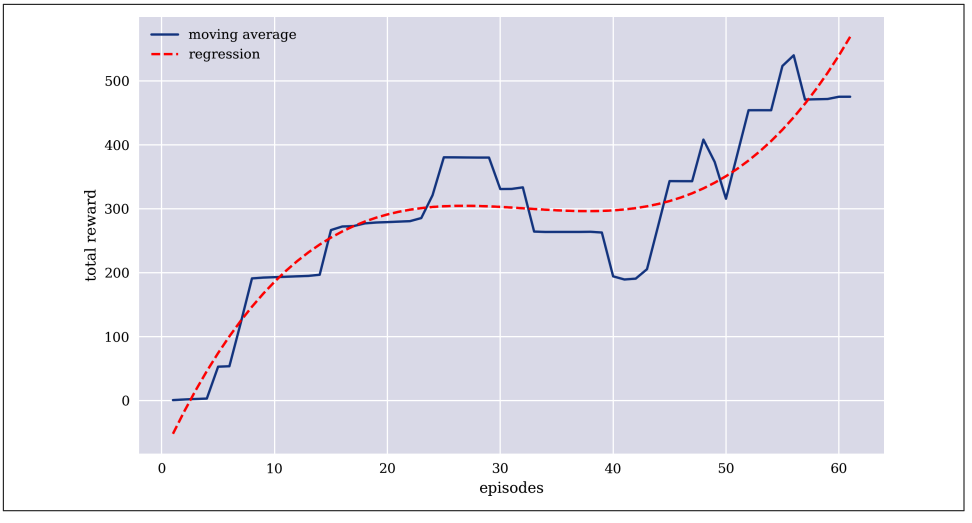

Figure 11-1. Average total reward per training episode

Figure 11-2 compares the gross performance of the trading bot on the training data—exhibiting quite some variance due to alternating between exploitation and exploration—with the one on the validation data set making use of exploitation only:

```
In [13]: tradingbot.plot_performance(agent)
```

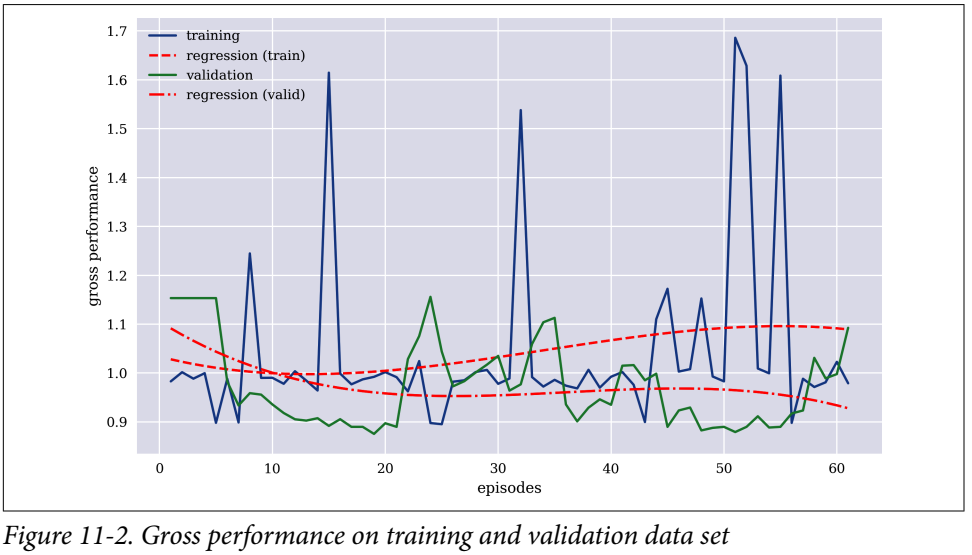

Figure 11-2. Gross performance on training and validation data set

This trained trading bot is used for backtesting in the following sections.

Vectorized Backtesting

Vectorized backtesting cannot directly be applied to the trading bot. Chapter 10 uses dense neural networks (DNNs) to illustrate the approach. In this context, the data with the features and labels sub-sets is prepared first and then fed to the DNN to generate all predictions at once. In a reinforcement learning (RL) context, data is generated and collected by interacting with the environment action by action and step by step.

To this end, the following Python code defines the `backtest` function, which takes as input a `TradingBot` instance and a `Finance` instance. It generates in the original `Data Frame` objects of the `Finance` environment columns with the positions the trading bot takes and the resulting strategy performance:

```
In [14]: def reshape(s):
             return np.reshape(s, [1, learn_env.lags,
                                   learn_env.n_features])  ❶

In [15]: def backtest(agent, env):
             env.min_accuracy = 0.0
             env.min_performance = 0.0
             done = False
             env.data['p'] = 0  ❷
             state = env.reset()
             while not done:
                 action = np.argmax(
                     agent.model.predict(reshape(state))[0, 0])  ❸
                 position = 1 if action == 1 else -1  ❹
                 env.data.loc[:, 'p'].iloc[env.bar] = position  ❺
                 state, reward, done, info = env.step(action)
             env.data['s'] = env.data['p'] * env.data['r'] * learn_env.leverage  ❻
```

❶ Reshapes a single feature-label combination

❷ Generates a column for the position values

❸ Derives the optimal action (prediction) given the trained DNN

❹ Derives the resulting position (`+1` for long/upwards, `-1` for short/downwards)…

❺ …and stores in the corresponding column at the appropriate index position

❻ Calculates the strategy log returns given the position values

Equipped with the `backtest` function, vectorized backtesting boils down to a few lines of Python code as in Chapter 10.

Figure 11-3 compares the passive benchmark investment's gross performance with the strategy gross performance:

```
In [16]: env = agent.learn_env  ❶

In [17]: backtest(agent, env)  ❷

In [18]: env.data['p'].iloc[env.lags:].value_counts()  ❸
Out[18]:  1    961
         -1    786
         Name: p, dtype: int64

In [19]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp)  ❹
Out[19]: r   0.7725
         s   1.5155
         dtype: float64

In [20]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp) - 1  ❺
Out[20]: r   -0.2275
         s    0.5155
         dtype: float64

In [21]: env.data[['r', 's']].iloc[env.lags:].cumsum(
                ).apply(np.exp).plot(figsize=(10, 6));
```

❶ Specifies the relevant environment

❷ Generates the additional data required

❸ Counts the number of long and short positions

❹ Calculates the gross performances for the passive benchmark investment (`r`) and the strategy (`s`)…

❺ …as well as the corresponding net performances

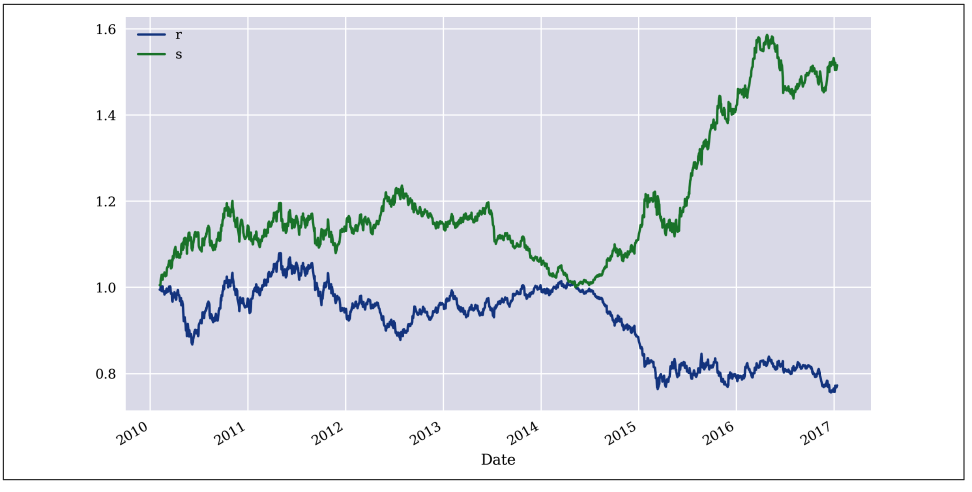

Figure 11-3. Gross performance of the passive benchmark investment and the trading bot (in-sample)

To get a more realistic picture of the performance of the trading bot, the following Python code creates a test environment with data that the trading bot has not yet seen. Figure 11-4 shows how the trading bot fares compared to the passive benchmark investment:

```
In [22]: test_env = finance.Finance(symbol, features=learn_env.features,
                                    window=learn_env.window,
                                    lags=learn_env.lags,
                                    leverage=learn_env.leverage,
                                    min_performance=0.0, min_accuracy=0.0,
                                    start=a + b + c, end=None,
                                    mu=learn_env.mu, std=learn_env.std)

In [23]: env = test_env

In [24]: backtest(agent, env)

In [25]: env.data['p'].iloc[env.lags:].value_counts()
Out[25]: -1    437
          1     56
         Name: p, dtype: int64

In [26]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp)
Out[26]: r   0.9144
         s   1.0992
         dtype: float64

In [27]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp) - 1
Out[27]: r   -0.0856
         s    0.0992
```

```
dtype: float64
```

```
In [28]: env.data[['r', 's']].iloc[env.lags:].cumsum(
                    ).apply(np.exp).plot(figsize=(10, 6));
```

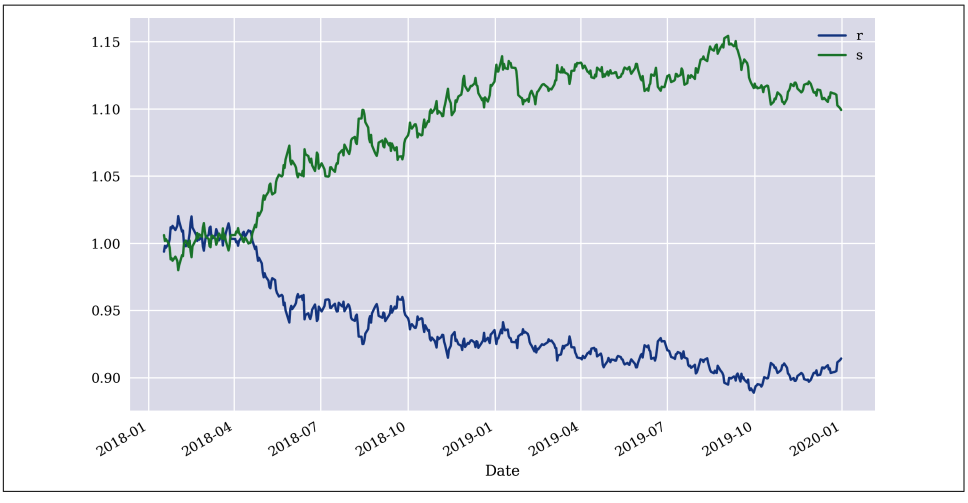

Figure 11-4. Gross performance of the passive benchmark investment and the trading bot (out-of-sample)

The out-of-sample performance without any risk measures implemented seems already promising. However, to be able to properly judge the real performance of a trading strategy, risk measures should be included. This is where event-based backtesting comes into play.

Event-Based Backtesting

Given the results of the previous section, the out-of-sample performance without any risk measures seems already promising. However, to be able to properly analyze risk measures, such as trailing stop loss orders, *event-based backtesting* is required. This section introduces this alternative approach to judging the performance of algorithmic trading strategies.

"Backtesting Base Class" on page 339 presents the `BacktestingBase` class that can be flexibly used to test different types of directional trading strategies. The code has detailed comments on the important lines. This base class provides the following methods:

`get_date_price()`
: For a given `bar` (index value for the `DataFrame` object containing the financial data), it returns the relevant `date` and `price`.

`print_balance()`
: For a given `bar`, it prints the current (cash) balance of the trading bot.

`calculate_net_wealth()`
: For a given `price`, it returns the net wealth composed of the current (cash) balance and the instrument position.

`print_net_wealth()`
: For a given `bar`, it prints the net wealth of the trading bot.

`place_buy_order()`, `place_sell_order()`
: For a given `bar` and a given number of `units` or a given `amount`, these methods place buy or sell orders and adjust relevant quantities accordingly (for example, accounting for transaction costs).

`close_out()`
: At a given `bar`, this method closes open positions and calculates and reports performance statistics.

The following Python code illustrates how an instance of the `BacktestingBase` class functions based on some simple steps:

```
In [29]: import backtesting as bt

In [30]: bb = bt.BacktestingBase(env=agent.learn_env, model=agent.model,
                                 amount=10000, ptc=0.0001, ftc=1.0,
                                 verbose=True)  ❶

In [31]: bb.initial_amount  ❷
Out[31]: 10000

In [32]: bar = 100  ❸

In [33]: bb.get_date_price(bar)  ❹
Out[33]: ('2010-06-25', 1.2374)

In [34]: bb.env.get_state(bar)  ❺
Out[34]:               EUR=       r       s       m      v
         Date
         2010-06-22 -0.0242 -0.5622 -0.0916 -0.2022 1.5316
         2010-06-23  0.0176  0.6940 -0.0939 -0.0915 1.5563
         2010-06-24  0.0354  0.3034 -0.0865  0.6391 1.0890

In [35]: bb.place_buy_order(bar, amount=5000)  ❻
         2010-06-25 | buy 4040 units for 1.2374
         2010-06-25 | current balance = 4999.40

In [36]: bb.print_net_wealth(2 * bar)  ❼
         2010-11-16 | net wealth = 10450.17
```

```
In [37]: bb.place_sell_order(2 * bar, units=1000)  ❽
         2010-11-16 | sell 1000 units for 1.3492
         2010-11-16 | current balance = 6347.47

In [38]: bb.close_out(3 * bar)  ❾
         ================================================
         2011-04-11 | *** CLOSING OUT ***
         2011-04-11 | sell 3040 units for 1.4434
         2011-04-11 | current balance = 10733.97
         2011-04-11 | net performance [%] = 7.3397
         2011-04-11 | number of trades [#] = 3
         ================================================
```

❶ Instantiates a `BacktestingBase` object

❷ Looks up the `initial_amount` attribute value

❸ Fixes a `bar` value

❹ Retrieves the `date` and `price` values for the `bar`

❺ Retrieves the state of the `Finance` environment for the `bar`

❻ Places a buy order using the `amount` parameter

❼ Prints the net wealth at a later point (`2 * bar`)

❽ Places a sell order at that later point using the `units` parameter

❾ Closes out the remaining long position even later (`3 * bar`)

Inheriting from the `BacktestingBase` class, the `TBBacktester` class implements the event-based backtesting for the trading bot:

```
In [39]: class TBBacktester(bt.BacktestingBase):
             def _reshape(self, state):
                 ''' Helper method to reshape state objects.
                 '''
                 return np.reshape(state, [1, self.env.lags, self.env.n_features])
             def backtest_strategy(self):
                 ''' Event-based backtesting of the trading bot's performance.
                 '''
                 self.units = 0
                 self.position = 0
                 self.trades = 0
                 self.current_balance = self.initial_amount
                 self.net_wealths = list()
                 for bar in range(self.env.lags, len(self.env.data)):
                     date, price = self.get_date_price(bar)
```

```
            if self.trades == 0:
                print(50 * '=')
                print(f'{date} | *** START BACKTEST ***')
                self.print_balance(bar)
                print(50 * '=')
            state = self.env.get_state(bar)  ❶
            action = np.argmax(self.model.predict(
                        self._reshape(state.values))[0, 0])  ❷
            position = 1 if action == 1 else -1  ❸
            if self.position in [0, -1] and position == 1:  ❹
                if self.verbose:
                    print(50 * '-')
                    print(f'{date} | *** GOING LONG ***')
                if self.position == -1:
                    self.place_buy_order(bar - 1, units=-self.units)
                self.place_buy_order(bar - 1,
                                     amount=self.current_balance)
                if self.verbose:
                    self.print_net_wealth(bar)
                self.position = 1
            elif self.position in [0, 1] and position == -1:  ❺
                if self.verbose:
                    print(50 * '-')
                    print(f'{date} | *** GOING SHORT ***')
                if self.position == 1:
                    self.place_sell_order(bar - 1, units=self.units)
                self.place_sell_order(bar - 1,
                                      amount=self.current_balance)
                if self.verbose:
                    self.print_net_wealth(bar)
                self.position = -1
            self.net_wealths.append((date,
                                     self.calculate_net_wealth(price)))  ❻
        self.net_wealths = pd.DataFrame(self.net_wealths,
                                        columns=['date', 'net_wealth'])  ❻
        self.net_wealths.set_index('date', inplace=True)  ❻
        self.net_wealths.index = pd.DatetimeIndex(
                                        self.net_wealths.index)  ❻
        self.close_out(bar)
```

❶ Retrieves the state of the `Finance` environment

❷ Generates the optimal action (prediction) given the state and the `model` object

❸ Derives the optimal position (long/short) given the optimal action (prediction)

❹ Enters a *long* position if the conditions are met

❺ Enters a *short* position if the conditions are met

❻ Collects the net wealth values over time and transforms them into a `DataFrame` object

The application of the `TBBacktester` class is straightforward, given that the `Finance` and `TradingBot` instances are already available. The following code backtests the trading bot first on the *learning environment* data—without and with transaction costs. Figure 11-5 compares the two cases visually over time:

```
In [40]: env = learn_env

In [41]: tb = TBBacktester(env, agent.model, 10000,
                           0.0, 0, verbose=False)  ❶

In [42]: tb.backtest_strategy()  ❶
         ================================================
         2010-02-05 | *** START BACKTEST ***
         2010-02-05 | current balance = 10000.00
         ================================================
         ================================================
         2017-01-12 | *** CLOSING OUT ***
         2017-01-12 | current balance = 14601.85
         2017-01-12 | net performance [%] = 46.0185
         2017-01-12 | number of trades [#] = 828
         ================================================

In [43]: tb_ = TBBacktester(env, agent.model, 10000,
                            0.00012, 0.0, verbose=False)

In [44]: tb_.backtest_strategy()  ❷
         ================================================
         2010-02-05 | *** START BACKTEST ***
         2010-02-05 | current balance = 10000.00
         ================================================
         ================================================
         2017-01-12 | *** CLOSING OUT ***
         2017-01-12 | current balance = 13222.08
         2017-01-12 | net performance [%] = 32.2208
         2017-01-12 | number of trades [#] = 828
         ================================================

In [45]: ax = tb.net_wealths.plot(figsize=(10, 6))
         tb_.net_wealths.columns = ['net_wealth (after tc)']
         tb_.net_wealths.plot(ax=ax);
```

❶ Event-based backtest in-sample *without* transaction costs

❷ Event-based backtest in-sample *with* transaction costs

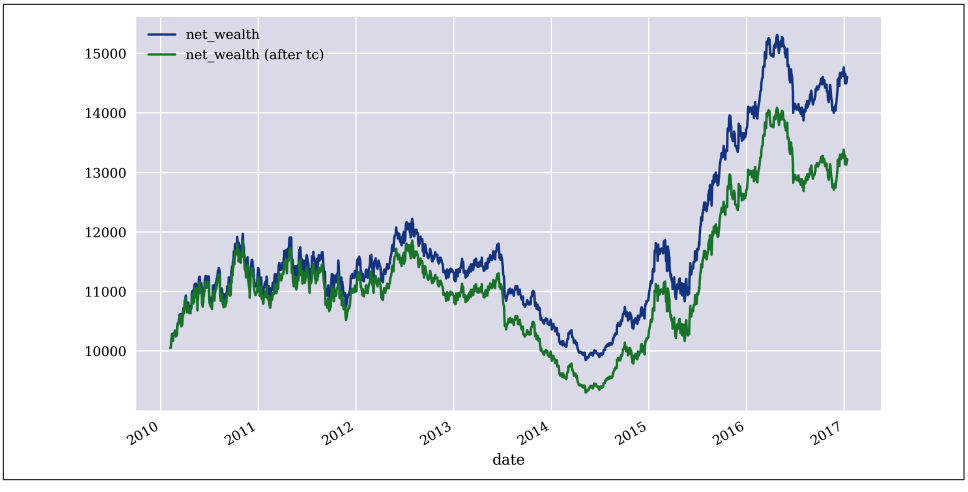

Figure 11-5. Gross performance of the trading bot before and after transaction costs (in-sample)

Figure 11-6 compares the gross performances of the trading bot for the *test environment* data over time—again, before and after transaction costs:

```
In [46]: env = test_env

In [47]: tb = TBBacktester(env, agent.model, 10000,
                           0.0, 0, verbose=False)  ❶

In [48]: tb.backtest_strategy()  ❶
         ================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ================================================
         ================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10936.79
         2019-12-31 | net performance [%] = 9.3679
         2019-12-31 | number of trades [#] = 186
         ================================================

In [49]: tb_ = TBBacktester(env, agent.model, 10000,
                            0.00012, 0.0, verbose=False)

In [50]: tb_.backtest_strategy()  ❷
         ================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ================================================
         ================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10695.72
```

```
2019-12-31 | net performance [%] = 6.9572
2019-12-31 | number of trades [#] = 186
================================================
```

```
In [51]: ax = tb.net_wealths.plot(figsize=(10, 6))
         tb_.net_wealths.columns = ['net_wealth (after tc)']
         tb_.net_wealths.plot(ax=ax);
```

❶ Event-based backtest out-of-sample *without* transaction costs

❷ Event-based backtest out-of-sample *with* transaction costs

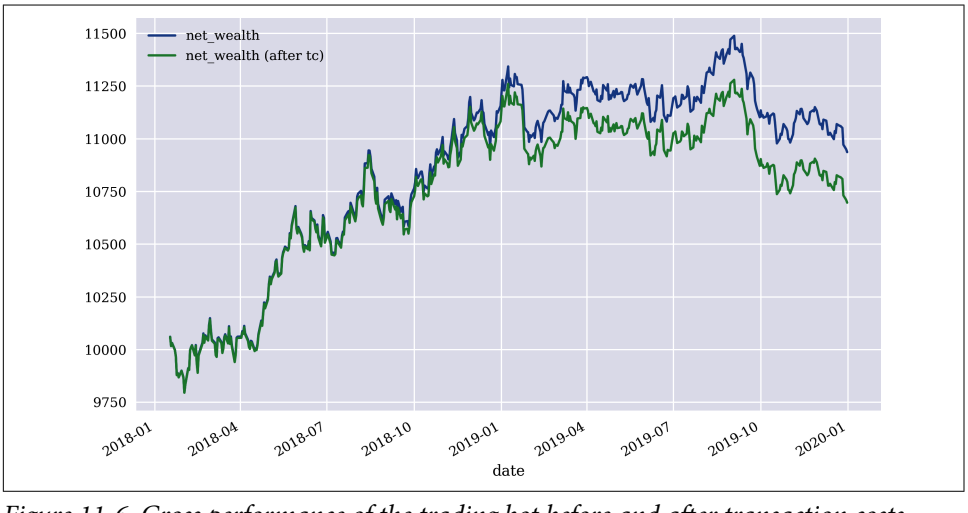

Figure 11-6. Gross performance of the trading bot before and after transaction costs (out-of-sample)

How does the performance before transaction costs from the event-based backtesting compare to the performance from the vectorized backtesting? Figure 11-7 shows the normalized net wealth compared to the gross performance over time. Due to the different technical approaches, the two time series are not exactly the same but are pretty similar. The performance difference can be mainly explained by the fact that the event-based backtesting assumes the same amount for every position taken. Vectorized backtesting takes compound effects into account, leading to a slightly higher reported performance:

```
In [52]: ax = (tb.net_wealths / tb.net_wealths.iloc[0]).plot(figsize=(10, 6))
         tp = env.data[['r', 's']].iloc[env.lags:].cumsum().apply(np.exp)
         (tp / tp.iloc[0]).plot(ax=ax);
```

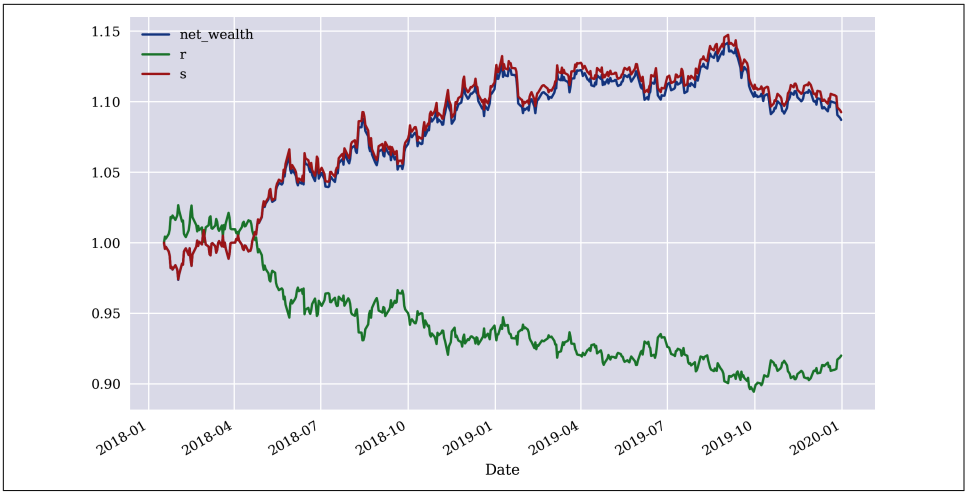

Figure 11-7. Gross performance of the passive benchmark investment and the trading bot (vectorized and event-based backtesting)

Performance Differences

The performance numbers from the vectorized and the event-based backtesting are close but not exactly the same. In the first case, it is assumed that financial instruments are perfectly divisible. Compounding is also done continuously. In the latter case, only full units of the financial instrument are accepted for trading, which is closer to reality. The net wealth calculations are based on price differences. The event-based code as it is used does not, for example, check whether the current balance is large enough to cover a certain trade by cash. This is for sure a simplifying assumption, and buying on margin, for instance, may not always be possible. Code adjustments in this regard are easily added to the `BacktestingBase` class.

Assessing Risk

The implementation of risk measures requires the understanding of the risks involved in trading the chosen financial instrument. Therefore, to properly set parameters for risk measures, such as stop loss orders, an assessment of the risk of the underlying instrument is important. There are many approaches available to measure the risk of a financial instrument. There are, for example, *nondirected risk measures*, such as volatility or average true range (ATR). There are also *directed measures*, such as maximum drawdown or value-at-risk (VaR).

A common practice when setting target levels for stop loss (SL), trailing stop loss (TSL), or take profit orders (TP) is to relate such levels to ATR values.[1] The following Python code calculates the ATR in absolute and relative terms for the financial instrument on which the trading bot is trained and backtested (that is, the EUR/USD exchange rate). The calculations rely on the data from the learning environment and use a typical window length of 14 days (bars). Figure 11-8 shows the calculated values, which vary significantly over time:

```
In [53]: data = pd.DataFrame(learn_env.data[symbol])  ❶

In [54]: data.head()  ❶
Out[54]:                EUR=
         Date
         2010-02-02 1.3961
         2010-02-03 1.3898
         2010-02-04 1.3734
         2010-02-05 1.3662
         2010-02-08 1.3652

In [55]: window = 14  ❷

In [56]: data['min'] = data[symbol].rolling(window).min()  ❸

In [57]: data['max'] = data[symbol].rolling(window).max()  ❹

In [58]: data['mami'] = data['max'] - data['min']  ❺

In [59]: data['mac'] = abs(data['max'] - data[symbol].shift(1))  ❻

In [60]: data['mic'] = abs(data['min'] - data[symbol].shift(1))  ❼

In [61]: data['atr'] = np.maximum(data['mami'], data['mac'])  ❽

In [62]: data['atr'] = np.maximum(data['atr'], data['mic'])  ❾

In [63]: data['atr%'] = data['atr'] / data[symbol]  ❿

In [64]: data[['atr', 'atr%']].plot(subplots=True, figsize=(10, 6));
```

❶ The instrument price column from the original `DataFrame` object

❷ The window length to be used for the calculations

❸ The rolling minimum

❹ The rolling maximum

1 For more details on the ATR measure, see ATR (1) Investopedia or ATR (2) Investopedia.

❺ The difference between rolling maximum and minimum

❻ The absolute difference between rolling maximum and previous day's price

❼ The absolute difference between rolling minimum and previous day's price

❽ The maximum of the max-min difference and the max-price difference

❾ The maximum between the previous maximum and the min-price difference (= ATR)

❿ The ATR value in percent from the absolute ATR value and the price

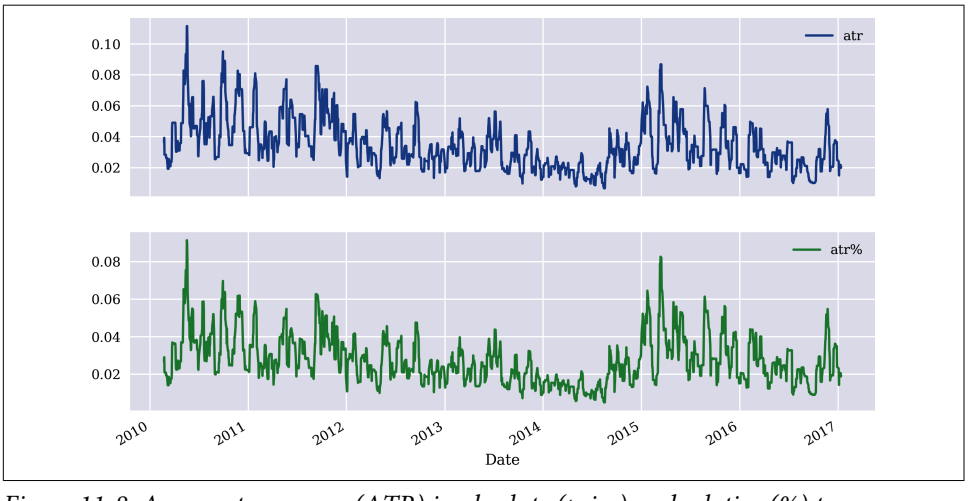

Figure 11-8. Average true range (ATR) in absolute (price) and relative (%) terms

The code that follows displays the final values for ATR in absolute and relative terms. A typical rule would be to set, for example, the SL level at the entry price minus x times ATR. Depending on the risk appetite of the trader or investor, x might be smaller than 1 or larger. This is where human judgment or formal risk policies come into play. If $x = 1$, then the SL level is set at about 2% below the entry level:

```
In [65]: data[['atr', 'atr%']].tail()
Out[65]:                atr    atr%
         Date
         2017-01-06 0.0218 0.0207
         2017-01-09 0.0218 0.0206
         2017-01-10 0.0218 0.0207
         2017-01-11 0.0199 0.0188
         2017-01-12 0.0206 0.0194
```

However, *leverage* plays an important role in this context. If a leverage of, say, 10 is used, which is actually quite low for foreign exchange trading, then the ATR numbers need to be multiplied by the leverage. As a consequence, for an assumed ATR factor of 1, the same SL level from before now is to be set at about 20% instead of just 2%. Or, when taking the median value of the ATR from the whole data set, it is set to be at about 25%:

```
In [66]: leverage = 10

In [67]: data[['atr', 'atr%']].tail() * leverage
Out[67]:                atr   atr%
         Date
         2017-01-06 0.2180 0.2070
         2017-01-09 0.2180 0.2062
         2017-01-10 0.2180 0.2066
         2017-01-11 0.1990 0.1881
         2017-01-12 0.2060 0.1942

In [68]: data[['atr', 'atr%']].median() * leverage
Out[68]: atr    0.3180
         atr%   0.2481
         dtype: float64
```

The basic idea behind relating SL or TP levels to ATR is that one should avoid setting them either too low or too high. Consider a 10 times leveraged position for which the ATR is 20%. Setting an SL level of only 3% or 5% might reduce the financial risk for the position, but it introduces the risk of a stop out that happens too early and that is due to typical movements in the financial instrument. Such "typical movements" within certain ranges are often called *noise*. The SL order should protect, in general, from unfavorable market movements that are larger than typical price movements (noise).

The same holds true for a take profit level. If it is set too high, say at three times the ATR level, decent profits might not be secured and positions might remain open for too long until they give up previous profits. Even if formal analyses and mathematical formulas can be used in this context, the setting of such target levels involves, as they say, more art than science. In a financial context, there is quite a degree of freedom for setting such target levels, and human judgment can come to the rescue. In other contexts, such as for AVs, this is different, as no human judgment is needed to instruct the AI to avoid any collisions with human beings.

NonNormality and NonLinearity

A *margin stop out* closes a trading position in cases when the margin, or the invested equity, is used up. Assume a leveraged trading position with a margin stop out in place. For a leverage of 10, for example, the margin is 10% equity. An *unfavorable* move of 10% or larger in the traded instrument eats up all the equity and triggers the close out of the position—a loss of 100% of the equity. A *favorable* move of the underlying of, say, 25% leads to a return on equity of 150%. Even if returns of the traded instrument are normally distributed, leverage and margin stop outs lead to nonnormally distributed returns and asymmetric, nonlinear relationships between the traded instrument and the trading position.

Backtesting Risk Measures

Having an idea of the ATR of a financial instrument is often a good start for the implementation of risk measures. To be able to properly backtest the effect of the typical risk management orders, some adjustments to the `BacktestingBase` class are helpful. The following Python code presents a new base class—`BacktestBaseRM`, which inherits from `BacktestingBase`—that helps in tracking the entry price of the previous trade as well as the maximum and minimum prices since that trade. These values are used to calculate the relevant performance measures during the event-based backtesting to which SL, TSL, and TP orders relate:

```
#
# Event-Based Backtesting
# --Base Class (2)
#
# (c) Dr. Yves J. Hilpisch
#
from backtesting import *


class BacktestingBaseRM(BacktestingBase):

    def set_prices(self, price):
        ''' Sets prices for tracking of performance.
            To test for e.g. trailing stop loss hit.
        '''
        self.entry_price = price  ❶
        self.min_price = price  ❷
        self.max_price = price  ❸

    def place_buy_order(self, bar, amount=None, units=None, gprice=None):
        ''' Places a buy order for a given bar and for
            a given amount or number of units.
        '''
        date, price = self.get_date_price(bar)
```

```
        if gprice is not None:
            price = gprice
        if units is None:
            units = int(amount / price)
        self.current_balance -= (1 + self.ptc) * units * price + self.ftc
        self.units += units
        self.trades += 1
        self.set_prices(price)  ❹
        if self.verbose:
            print(f'{date} | buy {units} units for {price:.4f}')
            self.print_balance(bar)

    def place_sell_order(self, bar, amount=None, units=None, gprice=None):
        ''' Places a sell order for a given bar and for
            a given amount or number of units.
        '''
        date, price = self.get_date_price(bar)
        if gprice is not None:
            price = gprice
        if units is None:
            units = int(amount / price)
        self.current_balance += (1 - self.ptc) * units * price - self.ftc
        self.units -= units
        self.trades += 1
        self.set_prices(price)  ❹
        if self.verbose:
            print(f'{date} | sell {units} units for {price:.4f}')
            self.print_balance(bar)
```

❶ Sets the *entry* price for the most recent trade

❷ Sets the initial *minimum* price since the most ecent trade

❸ Sets the initial *maximum* price since the most recent trade

❹ Sets the relevant prices after a trade is executed

Based on this new base class, "Backtesting Class" on page 342 presents a new backtesting class, `TBBacktesterRM`, that allows the inclusion of SL, TSL, and TP orders. The relevant code parts are discussed in the following sub-sections. The parametrization of the backtesting examples orients itself roughly on an ATR level of about 2%, as calculated in the previous section.

EUT and Risk Measures

EUT, MVP, and the CAPM (see Chapters 3 and 4) assume that financial agents know about the future distribution of the returns of a financial instrument. MPT and the CAPM assume furthermore that returns are normally distributed and that there is, for example, a linear relationship between the market portfolio's returns and the returns of a traded financial instrument. The use of SL, TSL, and TP orders leads—similar and in addition to leverage in combination with margin stop out—to a "guaranteed nonnormal" distribution and to highly asymmetric, nonlinear payoffs of a trading position in relation to the traded instrument.

Stop Loss

The first risk measure is the SL order. It fixes a certain price level or, more often, a fixed percent value that triggers the closing of a position. For example, if the entry price for an unleveraged position is 100 and the SL level is set to 5%, then a long position is closed out at 95 while a short position is closed out at 105.

The following Python code is the relevant part of the `TBBacktesterRM` class that handles an SL order. For the SL order, the class allows one to specify whether the price level for the order is guaranteed or not.[2] Working with guaranteed SL price levels might lead to too-optimistic performance results:

```
# stop loss order
if sl is not None and self.position != 0:  ❶
    rc = (price - self.entry_price) / self.entry_price  ❷
    if self.position == 1 and rc < -self.sl:  ❸
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 - self.sl)
            print(f'*** STOP LOSS (LONG  | {-self.sl:.4f}) ***')
        else:
            print(f'*** STOP LOSS (LONG  | {rc:.4f}) ***')
        self.place_sell_order(bar, units=self.units, gprice=price)  ❹
        self.wait = wait  ❺
        self.position = 0  ❻
    elif self.position == -1 and rc > self.sl:  ❼
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 + self.sl)
            print(f'*** STOP LOSS (SHORT | -{self.sl:.4f}) ***')
        else:
            print(f'*** STOP LOSS (SHORT | -{rc:.4f}) ***')
```

2 A *guaranteed* stop loss order might only be available in certain jurisdictions for certain groups of broker clients, such as retail investors/traders.

```
            self.place_buy_order(bar, units=-self.units, gprice=price)  ❽
            self.wait = wait  ❺
            self.position = 0  ❻
```

❶ Checks whether an SL is defined and whether the position is not neutral

❷ Calculates the performance based on the entry price for the last trade

❸ Checks whether an SL event is given for a *long* position

❹ Closes the *long* position, at either the current price or the guaranteed price level

❺ Sets the number of bars to wait before the next trade happens to `wait`

❻ Sets the position to neutral

❼ Checks whether an SL event is given for a *short* position

❽ Closes the *short* position, at either the current price or the guaranteed price level

The following Python code backtests the trading strategy of the trading bot without and with an SL order. For the given parametrization, the SL order has a negative impact on the strategy performance:

```
In [69]: import tbbacktesterrm as tbbrm

In [70]: env = test_env

In [71]: tb = tbbrm.TBBacktesterRM(env, agent.model, 10000,
                                   0.0, 0, verbose=False)  ❶

In [72]: tb.backtest_strategy(sl=None, tsl=None, tp=None, wait=5)  ❷
         ================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ================================================
         ================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10936.79
         2019-12-31 | net performance [%] = 9.3679
         2019-12-31 | number of trades [#] = 186
         ================================================

In [73]: tb.backtest_strategy(sl=0.0175, tsl=None, tp=None,
                              wait=5, guarantee=False)  ❸
         ================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ================================================
         ------------------------------------------------
```

```
*** STOP LOSS (SHORT | -0.0203) ***
================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10717.32
2019-12-31 | net performance [%] = 7.1732
2019-12-31 | number of trades [#] = 188
================================================

In [74]: tb.backtest_strategy(sl=0.017, tsl=None, tp=None,
                              wait=5, guarantee=True)  ❹
================================================
2018-01-17 | *** START BACKTEST ***
2018-01-17 | current balance = 10000.00
================================================
------------------------------------------------
*** STOP LOSS (SHORT | -0.0170) ***
================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10753.52
2019-12-31 | net performance [%] = 7.5352
2019-12-31 | number of trades [#] = 188
================================================
```

❶ Instantiates the backtesting class for risk management

❷ Backtests the trading bot performance without any risk measure

❸ Backtests the trading bot performance with an SL order (*no* guarantee)

❹ Backtests the trading bot performance with an SL order (*with* guarantee)

Trailing Stop Loss

In contrast to a regular SL order, a TSL order is adjusted whenever a new high is observed after the base order has been placed. Assume the base order for an unleveraged long position has an entry price of 95 and the TSL is set to 5%. If the instrument price reaches 100 and falls back to 95, this implies a TSL event, and the position is closed at the entry price level. If the price reaches 110 and falls back to 104.5, this would imply another TSL event.

The following Python code is the relevant part of the `TBBacktesterRM` class that handles a TSL order. To handle such an order correctly, the maximum prices (highs) and the minimum prices (lows) need to be tracked. The maximum price is relevant for a long position, whereas the minimum price is relevant for a short position:

```
# trailing stop loss order
if tsl is not None and self.position != 0:
    self.max_price = max(self.max_price, price)  ❶
    self.min_price = min(self.min_price, price)  ❷
```

```
        rc_1 = (price - self.max_price) / self.entry_price  ❸
        rc_2 = (self.min_price - price) / self.entry_price  ❹
        if self.position == 1 and rc_1 < -self.tsl:  ❺
            print(50 * '-')
            print(f'*** TRAILING SL (LONG  | {rc_1:.4f}) ***')
            self.place_sell_order(bar, units=self.units)
            self.wait = wait
            self.position = 0
        elif self.position == -1 and rc_2 < -self.tsl:  ❻
            print(50 * '-')
            print(f'*** TRAILING SL (SHORT | {rc_2:.4f}) ***')
            self.place_buy_order(bar, units=-self.units)
            self.wait = wait
            self.position = 0
```

❶ Updates the *maximum* price if necessary

❷ Updates the *minimum* price if necessary

❸ Calculates the relevant performance for a *long* position

❹ Calculates the relevant performance for a *short* position

❺ Checks whether a TSL event is given for a *long* position

❻ Checks whether a TSL event is given for a *short* position

As the backtesting results that follow show, using a TSL order with the given parametrization reduces the gross performance compared to a strategy without a TSL order in place:

```
In [75]: tb.backtest_strategy(sl=None, tsl=0.015,
                              tp=None, wait=5)  ❶
         ================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ================================================
         ------------------------------------------------
         *** TRAILING SL (SHORT | -0.0152) ***
         ------------------------------------------------
         *** TRAILING SL (SHORT | -0.0169) ***
         ------------------------------------------------
         *** TRAILING SL (SHORT | -0.0164) ***
         ------------------------------------------------
         *** TRAILING SL (SHORT | -0.0191) ***
         ------------------------------------------------
         *** TRAILING SL (SHORT | -0.0166) ***
         ------------------------------------------------
         *** TRAILING SL (SHORT | -0.0194) ***
         ------------------------------------------------
```

```
*** TRAILING SL (SHORT | -0.0172) ***
------------------------------------------------
*** TRAILING SL (SHORT | -0.0181) ***
------------------------------------------------
*** TRAILING SL (SHORT | -0.0153) ***
------------------------------------------------
*** TRAILING SL (SHORT | -0.0160) ***
================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10577.93
2019-12-31 | net performance [%] = 5.7793
2019-12-31 | number of trades [#] = 201
================================================
```

❶ Backtests the trading bot performance with a TSL order

Take Profit

Finally, there are TP orders. A TP order closes out a position that has reached a certain profit level. Say an unleveraged long position is opened at a price of 100 and the TP order is set to a level of 5%. If the price reaches 105, the position is closed.

The following code from the `TBBacktesterRM` class finally shows the part that handles a TP order. The TP implementation is straightforward, given the references of the SL and TSL order codes. For the TP order, there is also the option to backtest with a guaranteed price level as compared to the relevant high/low price levels, which would most probably lead to performance values that are too optimistic:[3]

```
# take profit order
if tp is not None and self.position != 0:
    rc = (price - self.entry_price) / self.entry_price
    if self.position == 1 and rc > self.tp:
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 + self.tp)
            print(f'*** TAKE PROFIT (LONG  | {self.tp:.4f}) ***')
        else:
            print(f'*** TAKE PROFIT (LONG  | {rc:.4f}) ***')
        self.place_sell_order(bar, units=self.units, gprice=price)
        self.wait = wait
        self.position = 0
    elif self.position == -1 and rc < -self.tp:
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 - self.tp)
            print(f'*** TAKE PROFIT (SHORT | {self.tp:.4f}) ***')
```

3 A take profit order has a fixed target price level. Therefore, it is unrealistic to use the high price of a time interval for a long position or the low price of the interval for a short position to calculate the realized profit.

```
        else:
            print(f'*** TAKE PROFIT (SHORT | {-rc:.4f}) ***')
        self.place_buy_order(bar, units=-self.units, gprice=price)
        self.wait = wait
        self.position = 0
```

For the given parametrization, adding a TP order—without guarantee—improves the trading bot performance noticeably compared to the passive benchmark investment. This result might be too optimistic given the considerations from before. Therefore, the TP order with guarantee leads to a more realistic performance value in this case:

```
In [76]: tb.backtest_strategy(sl=None, tsl=None, tp=0.015,
                              wait=5, guarantee=False)  ❶
         ================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ================================================
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0155) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0155) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0204) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0240) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0168) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0156) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0183) ***
         ================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 11210.33
         2019-12-31 | net performance [%] = 12.1033
         2019-12-31 | number of trades [#] = 198
         ================================================

In [77]: tb.backtest_strategy(sl=None, tsl=None, tp=0.015,
                              wait=5, guarantee=True)  ❷
         ================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ================================================
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         ------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
```

```
-------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0150) ***
-------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0150) ***
-------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0150) ***
=================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10980.86
2019-12-31 | net performance [%] = 9.8086
2019-12-31 | number of trades [#] = 198
=================================================
```

❶ Backtests the trading bot performance with a TP order (*no* guarantee)

❷ Backtests the trading bot performance with a TP order (*with* guarantee)

Of course, SL/TSL orders can also be combined with TP orders. The backtest results of the Python code that follows are in both cases worse than those for the strategy without the risk measures in place. In managing risk, there is hardly any free lunch:

```
In [78]: tb.backtest_strategy(sl=0.015, tsl=None,
                              tp=0.0185, wait=5)  ❶
         =================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         =================================================
         -------------------------------------------------
         *** STOP LOSS (SHORT | -0.0203) ***
         -------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0202) ***
         -------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0213) ***
         -------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0240) ***
         -------------------------------------------------
         *** STOP LOSS (SHORT | -0.0171) ***
         -------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0188) ***
         -------------------------------------------------
         *** STOP LOSS (SHORT | -0.0153) ***
         -------------------------------------------------
         *** STOP LOSS (SHORT | -0.0154) ***
         =================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10552.00
         2019-12-31 | net performance [%] = 5.5200
         2019-12-31 | number of trades [#] = 201
         =================================================

In [79]: tb.backtest_strategy(sl=None, tsl=0.02,
                              tp=0.02, wait=5)  ❷
```

```
================================================
2018-01-17 | *** START BACKTEST ***
2018-01-17 | current balance = 10000.00
================================================
------------------------------------------------
*** TRAILING SL (SHORT | -0.0235) ***
------------------------------------------------
*** TRAILING SL (SHORT | -0.0202) ***
------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0250) ***
------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0227) ***
------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0240) ***
------------------------------------------------
*** TRAILING SL (SHORT | -0.0216) ***
------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0241) ***
------------------------------------------------
*** TRAILING SL (SHORT | -0.0206) ***
================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10346.38
2019-12-31 | net performance [%] = 3.4638
2019-12-31 | number of trades [#] = 198
================================================
```

❶ Backtests the trading bot performance with an SL and TP order

❷ Backtests the trading bot performance with a TSL and TP order

Performance Impact

Risk measures have their reasoning and benefits. However, reducing risk may come at the price of lower overall performance. On the other hand, the backtesting example with the TP order shows performance improvements that can be explained by the fact that, given the ATR of a financial instrument, a certain profit level can be considered good enough to realize the profit. Any hope to see even higher profits typically is smashed by the market turning around again.

Conclusions

This chapter has three main topics. It backtests the performance of a trading bot (that is, a trained deep Q-learning agent) out-of-sample in both vectorized and event-based fashion. It also assesses risks in the form of the average true range (ATR) indicator that measures the *typical* variation in the price of the financial instrument of interest. Finally, the chapter discusses and backtests event-based typical risk measures in the form of stop loss (SL), trailing stop loss (TSL), and take profit (TP) orders.

Similar to autonomous vehicles (AVs), trading bots are hardly ever deployed based on the predictions of their AI only. To avoid large downside risks and to improve the (risk-adjusted) performance, risk measures usually come into play. Standard risk measures, as discussed in this chapter, are available on almost every trading platform, as well as for retail traders. The next chapter illustrates this in the context of the Oanda trading platform. The event-based backtesting approach provides the algorithmic flexibility to properly backtest the effects of such risk measures. While "reducing risk" may sound appealing, the backtest results indicate that the reduction in risk often comes at a cost: the performance might be lower when compared to the pure strategy without any risk measures. However, when finely tuned, the results also show that TP orders, for example, can also have a positive effect on the performance.

References

Books and papers cited in this chapter:

Agrawal, Ajay, Joshua Gans, and Avi Goldfarb. 2018. *Prediction Machines: The Simple Economics of Artificial Intelligence.* Boston: Harvard Business Review Press.

Hilpisch, Yves. 2020. *Python for Algorithmic Trading: From Idea to Cloud Deployment.* Sebastopol: O'Reilly.

Khonji, Majid, Jorge Dias, and Lakmal Seneviratne. 2019. "Risk-Aware Reasoning for Autonomous Vehicles." arXiv. October 6, 2019. *https://oreil.ly/2Z6WR*.

Python Code

Finance Environment

The following is the Python module with the `Finance` environment class:

```
#
# Finance Environment
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import math
import random
import numpy as np
import pandas as pd


class observation_space:
    def __init__(self, n):
        self.shape = (n,)


class action_space:
    def __init__(self, n):
        self.n = n

    def sample(self):
        return random.randint(0, self.n - 1)


class Finance:
    intraday = False
    if intraday:
        url = 'http://hilpisch.com/aiif_eikon_id_eur_usd.csv'
    else:
        url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'

    def __init__(self, symbol, features, window, lags,
                 leverage=1, min_performance=0.85, min_accuracy=0.5,
                 start=0, end=None, mu=None, std=None):
        self.symbol = symbol
        self.features = features
        self.n_features = len(features)
        self.window = window
        self.lags = lags
        self.leverage = leverage
        self.min_performance = min_performance
        self.min_accuracy = min_accuracy
        self.start = start
        self.end = end
```

```
        self.mu = mu
        self.std = std
        self.observation_space = observation_space(self.lags)
        self.action_space = action_space(2)
        self._get_data()
        self._prepare_data()

    def _get_data(self):
        self.raw = pd.read_csv(self.url, index_col=0,
                               parse_dates=True).dropna()
        if self.intraday:
            self.raw = self.raw.resample('30min', label='right').last()
            self.raw = pd.DataFrame(self.raw['CLOSE'])
            self.raw.columns = [self.symbol]

    def _prepare_data(self):
        self.data = pd.DataFrame(self.raw[self.symbol])
        self.data = self.data.iloc[self.start:]
        self.data['r'] = np.log(self.data / self.data.shift(1))
        self.data.dropna(inplace=True)
        self.data['s'] = self.data[self.symbol].rolling(self.window).mean()
        self.data['m'] = self.data['r'].rolling(self.window).mean()
        self.data['v'] = self.data['r'].rolling(self.window).std()
        self.data.dropna(inplace=True)
        if self.mu is None:
            self.mu = self.data.mean()
            self.std = self.data.std()
        self.data_ = (self.data - self.mu) / self.std
        self.data['d'] = np.where(self.data['r'] > 0, 1, 0)
        self.data['d'] = self.data['d'].astype(int)
        if self.end is not None:
            self.data = self.data.iloc[:self.end - self.start]
            self.data_ = self.data_.iloc[:self.end - self.start]

    def _get_state(self):
        return self.data_[self.features].iloc[self.bar -
                                              self.lags:self.bar]

    def get_state(self, bar):
        return self.data_[self.features].iloc[bar - self.lags:bar]

    def seed(self, seed):
        random.seed(seed)
        np.random.seed(seed)

    def reset(self):
        self.treward = 0
        self.accuracy = 0
        self.performance = 1
        self.bar = self.lags
        state = self.data_[self.features].iloc[self.bar -
                                               self.lags:self.bar]
```

```
        return state.values

    def step(self, action):
        correct = action == self.data['d'].iloc[self.bar]
        ret = self.data['r'].iloc[self.bar] * self.leverage
        reward_1 = 1 if correct else 0
        reward_2 = abs(ret) if correct else -abs(ret)
        self.treward += reward_1
        self.bar += 1
        self.accuracy = self.treward / (self.bar - self.lags)
        self.performance *= math.exp(reward_2)
        if self.bar >= len(self.data):
            done = True
        elif reward_1 == 1:
            done = False
        elif (self.performance < self.min_performance and
              self.bar > self.lags + 15):
            done = True
        elif (self.accuracy < self.min_accuracy and
              self.bar > self.lags + 15):
            done = True
        else:
            done = False
        state = self._get_state()
        info = {}
        return state.values, reward_1 + reward_2 * 5, done, info
```

Trading Bot

The following is the Python module with the `TradingBot` class, based on a financial Q-learning agent:

```
#
# Financial Q-Learning Agent
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import os
import random
import numpy as np
from pylab import plt, mpl
from collections import deque
import tensorflow as tf
from keras.layers import Dense, Dropout
from keras.models import Sequential
from keras.optimizers import Adam, RMSprop

os.environ['PYTHONHASHSEED'] = '0'
plt.style.use('seaborn')
mpl.rcParams['savefig.dpi'] = 300
mpl.rcParams['font.family'] = 'serif'
```

```
def set_seeds(seed=100):
    ''' Function to set seeds for all
        random number generators.
    '''
    random.seed(seed)
    np.random.seed(seed)
    tf.random.set_seed(seed)


class TradingBot:
    def __init__(self, hidden_units, learning_rate, learn_env,
                 valid_env=None, val=True, dropout=False):
        self.learn_env = learn_env
        self.valid_env = valid_env
        self.val = val
        self.epsilon = 1.0
        self.epsilon_min = 0.1
        self.epsilon_decay = 0.99
        self.learning_rate = learning_rate
        self.gamma = 0.5
        self.batch_size = 128
        self.max_treward = 0
        self.averages = list()
        self.trewards = []
        self.performances = list()
        self.aperformances = list()
        self.vperformances = list()
        self.memory = deque(maxlen=2000)
        self.model = self._build_model(hidden_units,
                                       learning_rate, dropout)

    def _build_model(self, hu, lr, dropout):
        ''' Method to create the DNN model.
        '''
        model = Sequential()
        model.add(Dense(hu, input_shape=(
            self.learn_env.lags, self.learn_env.n_features),
            activation='relu'))
        if dropout:
            model.add(Dropout(0.3, seed=100))
        model.add(Dense(hu, activation='relu'))
        if dropout:
            model.add(Dropout(0.3, seed=100))
        model.add(Dense(2, activation='linear'))
        model.compile(
            loss='mse',
            optimizer=RMSprop(lr=lr)
        )
        return model
```

```
    def act(self, state):
        ''' Method for taking action based on
            a) exploration
            b) exploitation
        '''
        if random.random() <= self.epsilon:
            return self.learn_env.action_space.sample()
        action = self.model.predict(state)[0, 0]
        return np.argmax(action)

    def replay(self):
        ''' Method to retrain the DNN model based on
            batches of memorized experiences.
        '''
        batch = random.sample(self.memory, self.batch_size)
        for state, action, reward, next_state, done in batch:
            if not done:
                reward += self.gamma * np.amax(
                    self.model.predict(next_state)[0, 0])
            target = self.model.predict(state)
            target[0, 0, action] = reward
            self.model.fit(state, target, epochs=1,
                           verbose=False)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def learn(self, episodes):
        ''' Method to train the DQL agent.
        '''
        for e in range(1, episodes + 1):
            state = self.learn_env.reset()
            state = np.reshape(state, [1, self.learn_env.lags,
                                       self.learn_env.n_features])
            for _ in range(10000):
                action = self.act(state)
                next_state, reward, done, info = self.learn_env.step(action)
                next_state = np.reshape(next_state,
                                        [1, self.learn_env.lags,
                                         self.learn_env.n_features])
                self.memory.append([state, action, reward,
                                    next_state, done])
                state = next_state
                if done:
                    treward = _ + 1
                    self.trewards.append(treward)
                    av = sum(self.trewards[-25:]) / 25
                    perf = self.learn_env.performance
                    self.averages.append(av)
                    self.performances.append(perf)
                    self.aperformances.append(
                        sum(self.performances[-25:]) / 25)
                    self.max_treward = max(self.max_treward, treward)
```

```
                    templ = 'episode: {:2d}/{} | treward: {:4d} | '
                    templ += 'perf: {:5.3f} | av: {:5.1f} | max: {:4d}'
                    print(templ.format(e, episodes, treward, perf,
                                       av, self.max_treward), end='\r')
                    break
            if self.val:
                self.validate(e, episodes)
            if len(self.memory) > self.batch_size:
                self.replay()
        print()

    def validate(self, e, episodes):
        ''' Method to validate the performance of the
            DQL agent.
        '''
        state = self.valid_env.reset()
        state = np.reshape(state, [1, self.valid_env.lags,
                                   self.valid_env.n_features])
        for _ in range(10000):
            action = np.argmax(self.model.predict(state)[0, 0])
            next_state, reward, done, info = self.valid_env.step(action)
            state = np.reshape(next_state, [1, self.valid_env.lags,
                                            self.valid_env.n_features])
            if done:
                treward = _ + 1
                perf = self.valid_env.performance
                self.vperformances.append(perf)
                if e % int(episodes / 6) == 0:
                    templ = 71 * '='
                    templ += '\nepisode: {:2d}/{} | VALIDATION | '
                    templ += 'treward: {:4d} | perf: {:5.3f} | eps: {:.2f}\n'
                    templ += 71 * '='
                    print(templ.format(e, episodes, treward,
                                       perf, self.epsilon))
                break


def plot_treward(agent):
    ''' Function to plot the total reward
        per training episode.
    '''
    plt.figure(figsize=(10, 6))
    x = range(1, len(agent.averages) + 1)
    y = np.polyval(np.polyfit(x, agent.averages, deg=3), x)
    plt.plot(x, agent.averages, label='moving average')
    plt.plot(x, y, 'r--', label='regression')
    plt.xlabel('episodes')
    plt.ylabel('total reward')
    plt.legend()


def plot_performance(agent):
```

```
    ''' Function to plot the financial gross
        performance per training episode.
    '''
    plt.figure(figsize=(10, 6))
    x = range(1, len(agent.performances) + 1)
    y = np.polyval(np.polyfit(x, agent.performances, deg=3), x)
    plt.plot(x, agent.performances[:], label='training')
    plt.plot(x, y, 'r--', label='regression (train)')
    if agent.val:
        y_ = np.polyval(np.polyfit(x, agent.vperformances, deg=3), x)
        plt.plot(x, agent.vperformances[:], label='validation')
        plt.plot(x, y_, 'r-.', label='regression (valid)')
    plt.xlabel('episodes')
    plt.ylabel('gross performance')
    plt.legend()
```

Backtesting Base Class

The following is the Python module with the `BacktestingBase` class for event-based backtesting:

```
#
# Event-Based Backtesting
# --Base Class (1)
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#


class BacktestingBase:
    def __init__(self, env, model, amount, ptc, ftc, verbose=False):
        self.env = env  ❶
        self.model = model  ❷
        self.initial_amount = amount  ❸
        self.current_balance = amount  ❸
        self.ptc = ptc  ❹
        self.ftc = ftc  ❺
        self.verbose = verbose  ❻
        self.units = 0  ❼
        self.trades = 0  ❽

    def get_date_price(self, bar):
        ''' Returns date and price for a given bar.
        '''
        date = str(self.env.data.index[bar])[:10]  ❾
        price = self.env.data[self.env.symbol].iloc[bar]  ❿
        return date, price

    def print_balance(self, bar):
        ''' Prints the current cash balance for a given bar.
        '''
```

```
        date, price = self.get_date_price(bar)
        print(f'{date} | current balance = {self.current_balance:.2f}')  ⓫

    def calculate_net_wealth(self, price):
        return self.current_balance + self.units * price  ⓬

    def print_net_wealth(self, bar):
        ''' Prints the net wealth for a given bar
            (cash + position).
        '''
        date, price = self.get_date_price(bar)
        net_wealth = self.calculate_net_wealth(price)
        print(f'{date} | net wealth = {net_wealth:.2f}')  ⓭

    def place_buy_order(self, bar, amount=None, units=None):
        ''' Places a buy order for a given bar and for
            a given amount or number of units.
        '''
        date, price = self.get_date_price(bar)
        if units is None:
            units = int(amount / price)  ⓮
            # units = amount / price  ⓮
        self.current_balance -= (1 + self.ptc) * \
            units * price + self.ftc  ⓯
        self.units += units  ⓰
        self.trades += 1  ⓱
        if self.verbose:
            print(f'{date} | buy {units} units for {price:.4f}')
            self.print_balance(bar)

    def place_sell_order(self, bar, amount=None, units=None):
        ''' Places a sell order for a given bar and for
            a given amount or number of units.
        '''
        date, price = self.get_date_price(bar)
        if units is None:
            units = int(amount / price)  ⓮
            # units = amount / price  ⓮
        self.current_balance += (1 - self.ptc) * \
            units * price - self.ftc  ⓯
        self.units -= units  ⓰
        self.trades += 1  ⓱
        if self.verbose:
            print(f'{date} | sell {units} units for {price:.4f}')
            self.print_balance(bar)

    def close_out(self, bar):
        ''' Closes out any open position at a given bar.
        '''
        date, price = self.get_date_price(bar)
        print(50 * '=')
        print(f'{date} | *** CLOSING OUT ***')
```

```
        if self.units < 0:
            self.place_buy_order(bar, units=-self.units)  ⓲
        else:
            self.place_sell_order(bar, units=self.units)  ⓳
        if not self.verbose:
            print(f'{date} | current balance = {self.current_balance:.2f}')
        perf = (self.current_balance / self.initial_amount - 1) * 100  ⓴
        print(f'{date} | net performance [%] = {perf:.4f}')
        print(f'{date} | number of trades [#] = {self.trades}')
        print(50 * '=')
```

❶ The relevant `Finance` environment

❷ The relevant DNN model (from the trading bot)

❸ The initial/current balance

❹ Proportional transaction costs

❺ Fixed transaction costs

❻ Whether the prints are verbose or not

❼ The initial number of units of the financial instrument traded

❽ The initial number of trades implemented

❾ The relevant *date* given a certain bar

❿ The relevant *instrument price* at a certain bar

⓫ The output of the *date* and *current balance* for a certain bar

⓬ The calculation of the *net wealth* from the current balance and the instrument position

⓭ The output of the *date* and the *net wealth* at a certain bar

⓮ The number of units to be traded given the trade amount

⓯ The impact of the trade and the associated costs on the current balance

⓰ The adjustment of the number of units held

⓱ The adjustment of the number of trades implemented

⓲ The closing of a *short* position…

⓳ …or of a *long* position

⓴ The net performance given the initial amount and the final current balance

Backtesting Class

The following is the Python module with the `TBBacktesterRM` class for event-based backtesting including risk measures (stop loss, trailing stop loss, take profit orders):

```
#
# Event-Based Backtesting
# --Trading Bot Backtester
# (incl. Risk Management)
#
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd
import backtestingrm as btr


class TBBacktesterRM(btr.BacktestingBaseRM):
    def _reshape(self, state):
        ''' Helper method to reshape state objects.
        '''
        return np.reshape(state, [1, self.env.lags, self.env.n_features])

    def backtest_strategy(self, sl=None, tsl=None, tp=None,
                          wait=5, guarantee=False):
        ''' Event-based backtesting of the trading bot's performance.
            Incl. stop loss, trailing stop loss and take profit.
        '''
        self.units = 0
        self.position = 0
        self.trades = 0
        self.sl = sl
        self.tsl = tsl
        self.tp = tp
        self.wait = 0
        self.current_balance = self.initial_amount
        self.net_wealths = list()
        for bar in range(self.env.lags, len(self.env.data)):
            self.wait = max(0, self.wait - 1)
            date, price = self.get_date_price(bar)
            if self.trades == 0:
                print(50 * '=')
                print(f'{date} | *** START BACKTEST ***')
                self.print_balance(bar)
                print(50 * '=')
```

```
        # stop loss order
        if sl is not None and self.position != 0:
            rc = (price - self.entry_price) / self.entry_price
            if self.position == 1 and rc < -self.sl:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 - self.sl)
                    print(f'*** STOP LOSS (LONG  | {-self.sl:.4f}) ***')
                else:
                    print(f'*** STOP LOSS (LONG  | {rc:.4f}) ***')
                self.place_sell_order(bar, units=self.units, gprice=price)
                self.wait = wait
                self.position = 0
            elif self.position == -1 and rc > self.sl:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 + self.sl)
                    print(f'*** STOP LOSS (SHORT | -{self.sl:.4f}) ***')
                else:
                    print(f'*** STOP LOSS (SHORT | -{rc:.4f}) ***')
                self.place_buy_order(bar, units=-self.units, gprice=price)
                self.wait = wait
                self.position = 0

        # trailing stop loss order
        if tsl is not None and self.position != 0:
            self.max_price = max(self.max_price, price)
            self.min_price = min(self.min_price, price)
            rc_1 = (price - self.max_price) / self.entry_price
            rc_2 = (self.min_price - price) / self.entry_price
            if self.position == 1 and rc_1 < -self.tsl:
                print(50 * '-')
                print(f'*** TRAILING SL (LONG  | {rc_1:.4f}) ***')
                self.place_sell_order(bar, units=self.units)
                self.wait = wait
                self.position = 0
            elif self.position == -1 and rc_2 < -self.tsl:
                print(50 * '-')
                print(f'*** TRAILING SL (SHORT | {rc_2:.4f}) ***')
                self.place_buy_order(bar, units=-self.units)
                self.wait = wait
                self.position = 0

        # take profit order
        if tp is not None and self.position != 0:
            rc = (price - self.entry_price) / self.entry_price
            if self.position == 1 and rc > self.tp:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 + self.tp)
                    print(f'*** TAKE PROFIT (LONG  | {self.tp:.4f}) ***')
```

```
                else:
                    print(f'*** TAKE PROFIT (LONG  | {rc:.4f}) ***')
                self.place_sell_order(bar, units=self.units, gprice=price)
                self.wait = wait
                self.position = 0
            elif self.position == -1 and rc < -self.tp:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 - self.tp)
                    print(f'*** TAKE PROFIT (SHORT | {self.tp:.4f}) ***')
                else:
                    print(f'*** TAKE PROFIT (SHORT | {-rc:.4f}) ***')
                self.place_buy_order(bar, units=-self.units, gprice=price)
                self.wait = wait
                self.position = 0

        state = self.env.get_state(bar)
        action = np.argmax(self.model.predict(
            self._reshape(state.values))[0, 0])
        position = 1 if action == 1 else -1
        if self.position in [0, -1] and position == 1 and self.wait == 0:
            if self.verbose:
                print(50 * '-')
                print(f'{date} | *** GOING LONG ***')
            if self.position == -1:
                self.place_buy_order(bar - 1, units=-self.units)
            self.place_buy_order(bar - 1, amount=self.current_balance)
            if self.verbose:
                self.print_net_wealth(bar)
            self.position = 1
        elif self.position in [0, 1] and position == -1 and self.wait == 0:
            if self.verbose:
                print(50 * '-')
                print(f'{date} | *** GOING SHORT ***')
            if self.position == 1:
                self.place_sell_order(bar - 1, units=self.units)
            self.place_sell_order(bar - 1, amount=self.current_balance)
            if self.verbose:
                self.print_net_wealth(bar)
            self.position = -1
        self.net_wealths.append((date, self.calculate_net_wealth(price)))
    self.net_wealths = pd.DataFrame(self.net_wealths,
                                    columns=['date', 'net_wealth'])
    self.net_wealths.set_index('date', inplace=True)
    self.net_wealths.index = pd.DatetimeIndex(self.net_wealths.index)
    self.close_out(bar)
```

CHAPTER 12
Execution and Deployment

> Considerable progress is needed before autonomous vehicles can operate reliably in mixed urban traffic, heavy rain and snow, unpaved and unmapped roads, and where wireless access is unreliable.
>
> —Todd Litman (2020)

> An investment firm that engages in algorithmic trading shall have in place effective systems and risk controls suitable to the business it operates to ensure that its trading systems are resilient and have sufficient capacity, are subject to appropriate trading thresholds and limits and prevent the sending of erroneous orders or the systems otherwise functioning in a way that may create or contribute to a disorderly market.
>
> —MiFID II (Article 17)

Chapter 11 trains a trading bot in the form of a financial Q-learning agent based on historical data. It introduces event-based backtesting as an approach flexible enough to account for typical risk measures, such as trailing stop loss orders or take profit targets. However, all this happens asynchronously in a sandbox environment based on historical data only. As with an autonomous vehicle (AV), there is the problem of deploying the AI in the real world. For an AV this means combining the AI with the car hardware and deploying the AV on test and public streets. For a trading bot this means connecting the trading bot with a trading platform and deploying it such that orders are executed automatically. In other words, the algorithmic side is clear—execution and deployment now need to be added to implement algorithmic trading.

This chapter introduces the Oanda trading platform for algorithmic trading. Therefore, the focus is on the v20 API of the platform and not on applications that provide users with an interface for manual trading. To simplify the code, the wrapper package `tpqoa` is introduced and used. It relies on the `v20` Python package from Oanda and provides a more Pythonic user interface.

"Oanda Account" on page 346 details the prerequisites to use a *demo account* with Oanda. "Data Retrieval" on page 347 shows how to retrieve historical and real-time (streaming) data from the API. "Order Execution" on page 351 deals with the execution of buy and sell orders, potentially including other orders, such as trailing stop loss orders. "Trading Bot" on page 357 trains a trading bot based on historical intraday data from Oanda and backtests its performance in vectorized fashion. Finally, "Deployment" on page 364 shows how to deploy the trading bot in real-time and an automated fashion.

Oanda Account

The code in this chapter relies on the Python wrapper package `tpqoa`. This package can be installed via `pip` as follows:

```
pip install --upgrade git+https://github.com/yhilpisch/tpqoa.git
```

To make use of this package, a demo account with Oanda is sufficient. Once the account is open, an *access token* is generated on the account page (after login). The access token and the *account id* (also found on the account page) are then stored in a configuration text file as follows:

```
[oanda]
account_id = XYZ-ABC-...
access_token = ZYXCAB...
account_type = practice
```

If the name of the configuration file is *aiif.cfg* and if it is stored in the current working directory, then the `tpqoa` package can be used as follows:

```
import tpqoa
api = tpqoa.tpqoa('aiif.cfg')
```

Risk Disclaimers and Disclosures

Oanda is a platform for *foreign exchange* (FX) and *contracts for difference* (CFD) trading. These instruments involve considerable risks, in particular when traded with leverage. It is strongly recommended that you read all relevant risk disclaimers and disclosures from Oanda on its website carefully before moving on (check for the appropriate jurisdiction).

All code and examples presented in this chapter are for technical illustration only and do not constitute any investment advice or similar.

Data Retrieval

As usual, some Python imports and configurations come first:

```
In [1]: import os
        import time
        import numpy as np
        import pandas as pd
        from pprint import pprint
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('mode.chained_assignment', None)
        pd.set_option('display.float_format', '{:.5f}'.format)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'
```

Depending on the relevant jurisdiction of the account, Oanda offers a number of tradable FX and CFD instruments. The following Python code retrieves the available instruments for a given account:

```
In [2]: import tpqoa  ❶

In [3]: api = tpqoa.tpqoa('../aiif.cfg')  ❷

In [4]: ins = api.get_instruments()  ❸

In [5]: ins[:5]  ❹
Out[5]: [('AUD/CAD', 'AUD_CAD'),
         ('AUD/CHF', 'AUD_CHF'),
         ('AUD/HKD', 'AUD_HKD'),
         ('AUD/JPY', 'AUD_JPY'),
         ('AUD/NZD', 'AUD_NZD')]
```

❶ Imports the `tpqoa` package

❷ Instantiates an API object given the account credentials

❸ Retrieves the list of available instruments in the format `(display_name, technical_name)`

❹ Shows a select few of these instruments

Oanda provides a wealth of historical data via its v20 API. The following examples retrieve historical data for the EUR/USD currency pair—the granularity is set to `D` (that is, *daily*).

Figure 12-1 plots the closing (ask) prices:

```
In [6]: raw = api.get_history(instrument='EUR_USD',  ❶
                              start='2018-01-01',  ❷
                              end='2020-07-31',  ❸
                              granularity='D',  ❹
                              price='A')  ❺

In [7]: raw.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 671 entries, 2018-01-01 22:00:00 to 2020-07-30 21:00:00
        Data columns (total 6 columns):
         #   Column    Non-Null Count  Dtype
        ---  ------    --------------  -----
         0   o         671 non-null    float64
         1   h         671 non-null    float64
         2   l         671 non-null    float64
         3   c         671 non-null    float64
         4   volume    671 non-null    int64
         5   complete  671 non-null    bool
        dtypes: bool(1), float64(4), int64(1)
        memory usage: 32.1 KB

In [8]: raw.head()
Out[8]:                          o       h       l       c  volume  complete
        time
        2018-01-01 22:00:00 1.20101 1.20819 1.20051 1.20610   35630      True
        2018-01-02 22:00:00 1.20620 1.20673 1.20018 1.20170   31354      True
        2018-01-03 22:00:00 1.20170 1.20897 1.20049 1.20710   35187      True
        2018-01-04 22:00:00 1.20692 1.20847 1.20215 1.20327   36478      True
        2018-01-07 22:00:00 1.20301 1.20530 1.19564 1.19717   27618      True

In [9]: raw['c'].plot(figsize=(10, 6));
```

❶ Specifies the instrument…

❷ …the starting date…

❸ …the end date…

❹ …the granularity (`D` = daily)…

❺ …and the type of the price series (`A` = ask)

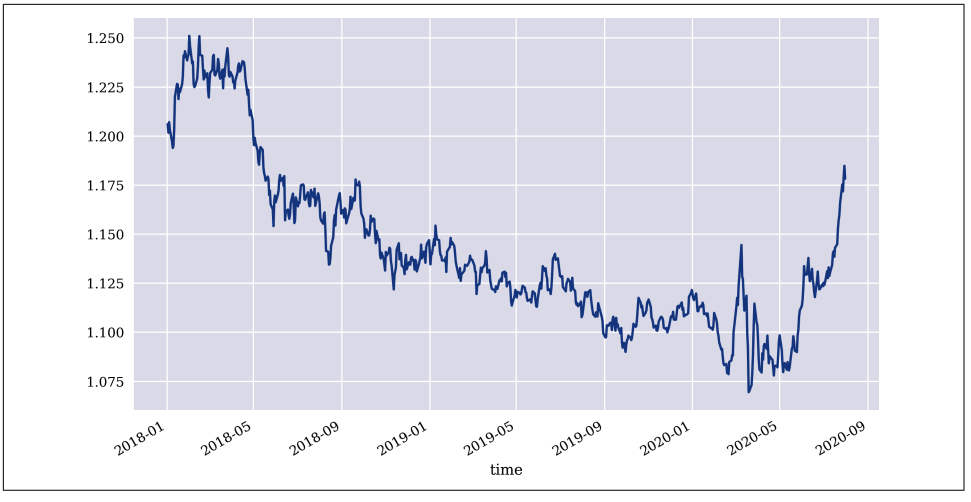

Figure 12-1. Historical daily closing prices for EUR/USD from Oanda

Intraday data is as easily retrieved and used as daily data, as the code that follows shows. Figure 12-2 visualizes minute bar (mid) price data:

```
In [10]: raw = api.get_history(instrument='EUR_USD',
                               start='2020-07-01',
                               end='2020-07-31',
                               granularity='M1',  ❶
                               price='M')  ❷

In [11]: raw.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 30728 entries, 2020-07-01 00:00:00 to 2020-07-30 23:59:00
         Data columns (total 6 columns):
          #   Column    Non-Null Count  Dtype
         ---  ------    --------------  -----
          0   o         30728 non-null  float64
          1   h         30728 non-null  float64
          2   l         30728 non-null  float64
          3   c         30728 non-null  float64
          4   volume    30728 non-null  int64
          5   complete  30728 non-null  bool
         dtypes: bool(1), float64(4), int64(1)
         memory usage: 1.4 MB

In [12]: raw.tail()
Out[12]:                          o       h       l       c  volume  complete
         time
         2020-07-30 23:55:00 1.18724 1.18739 1.18718 1.18738      57      True
         2020-07-30 23:56:00 1.18736 1.18758 1.18722 1.18757      57      True
         2020-07-30 23:57:00 1.18756 1.18756 1.18734 1.18734      49      True
         2020-07-30 23:58:00 1.18736 1.18737 1.18713 1.18717      36      True
         2020-07-30 23:59:00 1.18718 1.18724 1.18714 1.18722      31      True
```

```
In [13]: raw['c'].plot(figsize=(10, 6));
```

❶ Specifies the granularity (`M1` = one minute)…

❷ …and the type of the price series (`M` = mid)

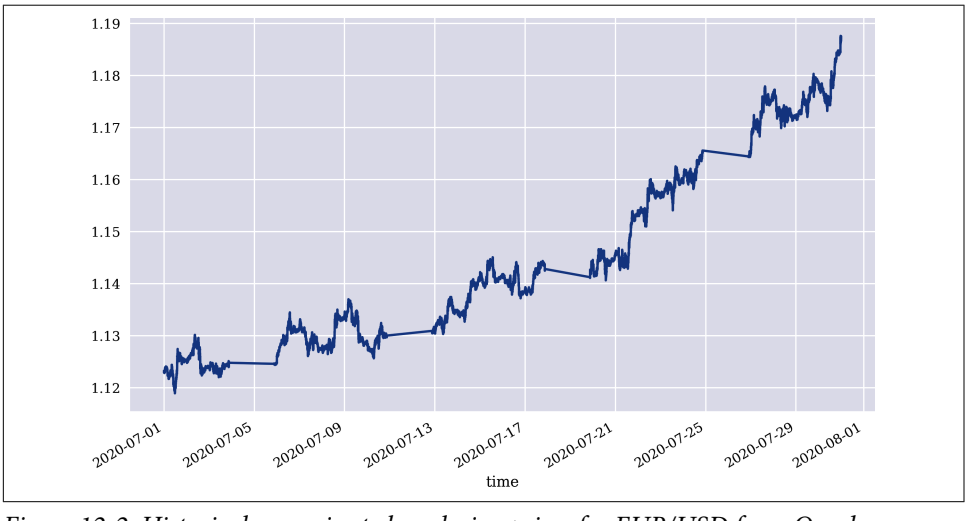

Figure 12-2. Historical one-minute bar closing prices for EUR/USD from Oanda

Whereas historical data is important, for instance, to train and test a trading bot, real-time (streaming) data is required to deploy such a bot for algorithmic trading. `tpqoa` allows the synchronous streaming of real-time data for all available instruments with a single method call. The method prints by default the time stamp and the bid/ask prices. For algorithmic trading, this default behavior can be adjusted, as "Deployment" on page 364 shows:

```
In [14]: api.stream_data('EUR_USD', stop=10)
         2020-08-13T12:07:09.735715316Z 1.18328 1.18342
         2020-08-13T12:07:16.245253689Z 1.18329 1.18343
         2020-08-13T12:07:16.397803785Z 1.18328 1.18342
         2020-08-13T12:07:17.240232521Z 1.18331 1.18346
         2020-08-13T12:07:17.358476854Z 1.18334 1.18348
         2020-08-13T12:07:17.778061207Z 1.18331 1.18345
         2020-08-13T12:07:18.016544856Z 1.18333 1.18346
         2020-08-13T12:07:18.144762415Z 1.18334 1.18348
         2020-08-13T12:07:18.689365678Z 1.18331 1.18345
         2020-08-13T12:07:19.148039139Z 1.18331 1.18345
```

Order Execution

The AI of an AV needs to be able to control the physical vehicle. To this end it sends different types of signals to the vehicle, for example, to accelerate, break, turn left, or turn right. A trading bot needs to be able to place orders with the trading platform. This section covers different types of orders, such as market orders and stop loss orders.

The most fundamental type of order is a *market order*. This order allows buying or selling a financial instrument at the current market price (that is, the *ask price* when buying and the *bid price* when selling). The following examples are based on an account leverage of 20 and relatively small order sizes. Therefore, liquidity issues, for example, do not play a role. When executing orders via the Oanda v20 API, the API returns a detailed order object. First, a *buy market order* is placed:

```
In [15]: order = api.create_order('EUR_USD', units=25000,
                                  suppress=True, ret=True)  ❶
         pprint(order)  ❶
         {'accountBalance': '98553.3172',
          'accountID': '101-004-13834683-001',
          'batchID': '1625',
          'commission': '0.0',
          'financing': '0.0',
          'fullPrice': {'asks': [{'liquidity': '10000000', 'price': 1.18345}],
                        'bids': [{'liquidity': '10000000', 'price': 1.18331}],
                        'closeoutAsk': 1.18345,
                        'closeoutBid': 1.18331,
                        'type': 'PRICE'},
          'fullVWAP': 1.18345,
          'gainQuoteHomeConversionFactor': '0.840811914585',
          'guaranteedExecutionFee': '0.0',
          'halfSpreadCost': '1.4788',
          'id': '1626',
          'instrument': 'EUR_USD',
          'lossQuoteHomeConversionFactor': '0.849262285586',
          'orderID': '1625',
          'pl': '0.0',
          'price': 1.18345,
          'reason': 'MARKET_ORDER',
          'requestID': '78757241547812154',
          'time': '2020-08-13T12:07:19.434407966Z',
          'tradeOpened': {'guaranteedExecutionFee': '0.0',
                          'halfSpreadCost': '1.4788',
                          'initialMarginRequired': '832.5',
                          'price': 1.18345,
                          'tradeID': '1626',
                          'units': '25000.0'},
          'type': 'ORDER_FILL',
          'units': '25000.0',
          'userID': 13834683}
```

```
In [16]: def print_details(order):  ❷
             details = (order['time'][:-7], order['instrument'], order['units'],
                        order['price'], order['pl'])
             return details

In [17]: print_details(order)  ❷
Out[17]: ('2020-08-13T12:07:19.434', 'EUR_USD', '25000.0', 1.18345, '0.0')

In [18]: time.sleep(1)
```

❶ Places a *buy market order* and prints the order object details

❷ Selects and shows the `time`, `instrument`, `units`, `price`, and `pl` details of the order

Second, the position is closed via a *sell market order* of the same size. Whereas the first trade has a profit/loss (P&L) of zero by its nature—before accounting for transaction costs—the second trade in general has a nonzero P&L:

```
In [19]: order = api.create_order('EUR_USD', units=-25000,
                                  suppress=True, ret=True)  ❶
         pprint(order)  ❶
         {'accountBalance': '98549.283',
          'accountID': '101-004-13834683-001',
          'batchID': '1627',
          'commission': '0.0',
          'financing': '0.0',
          'fullPrice': {'asks': [{'liquidity': '9975000', 'price': 1.18339}],
                        'bids': [{'liquidity': '10000000', 'price': 1.18326}],
                        'closeoutAsk': 1.18339,
                        'closeoutBid': 1.18326,
                        'type': 'PRICE'},
          'fullVWAP': 1.18326,
          'gainQuoteHomeConversionFactor': '0.840850994445',
          'guaranteedExecutionFee': '0.0',
          'halfSpreadCost': '1.3732',
          'id': '1628',
          'instrument': 'EUR_USD',
          'lossQuoteHomeConversionFactor': '0.849301758209',
          'orderID': '1627',
          'pl': '-4.0342',
          'price': 1.18326,
          'reason': 'MARKET_ORDER',
          'requestID': '78757241552009237',
          'time': '2020-08-13T12:07:20.586564454Z',
          'tradesClosed': [{'financing': '0.0',
                            'guaranteedExecutionFee': '0.0',
                            'halfSpreadCost': '1.3732',
                            'price': 1.18326,
                            'realizedPL': '-4.0342',
                            'tradeID': '1626',
                            'units': '-25000.0'}],
```

```
          'type': 'ORDER_FILL',
          'units': '-25000.0',
          'userID': 13834683}

In [20]: print_details(order) ❷
Out[20]: ('2020-08-13T12:07:20.586', 'EUR_USD', '-25000.0', 1.18326, '-4.0342')

In [21]: time.sleep(1)
```

❶ Places a *sell market order* and prints the order object details

❷ Selects and shows the `time`, `instrument`, `units`, `price`, and `pl` details of the order

Limit Orders

This chapter covers *market orders* as a type of base order only. With a market order, buying or selling a financial instrument happens at the price that is current when the order is placed. By contrast, a *limit order*, as the other main type of base order, allows the placement of an order with a minimum price or a maximum price. Only when the minimum/maximum price is reached is the order executed. Until that point, no transaction takes place.

Next, consider an example for the same combination of trades but this time with a *stop loss* (SL) order. An SL order is treated as a separate (limit) order. The following Python code places the orders and shows the details of the SL order object:

```
In [22]: order = api.create_order('EUR_USD', units=25000,
                                  sl_distance=0.005,  ❶
                                  suppress=True, ret=True)

In [23]: print_details(order)
Out[23]: ('2020-08-13T12:07:21.740', 'EUR_USD', '25000.0', 1.18343, '0.0')

In [24]: sl_order = api.get_transaction(tid=int(order['id']) + 1)  ❷

In [25]: sl_order  ❷
Out[25]: {'id': '1631',
          'time': '2020-08-13T12:07:21.740825489Z',
          'userID': 13834683,
          'accountID': '101-004-13834683-001',
          'batchID': '1629',
          'requestID': '78757241556206373',
          'type': 'STOP_LOSS_ORDER',
          'tradeID': '1630',
          'price': 1.17843,
          'distance': '0.005',
          'timeInForce': 'GTC',
          'triggerCondition': 'DEFAULT',
          'reason': 'ON_FILL'}
```

```
In [26]: (sl_order['time'], sl_order['type'], order['price'],
          sl_order['price'], sl_order['distance'])  ❸
Out[26]: ('2020-08-13T12:07:21.740825489Z',
          'STOP_LOSS_ORDER',
          1.18343,
          1.17843,
          '0.005')

In [27]: time.sleep(1)

In [28]: order = api.create_order('EUR_USD', units=-25000, suppress=True, ret=True)

In [29]: print_details(order)
Out[29]: ('2020-08-13T12:07:23.059', 'EUR_USD', '-25000.0', 1.18329, '-2.9725')
```

❶ The SL distance is defined in currency units.

❷ Selects and shows the SL order object data.

❸ Selects and shows some relevant details of the two order objects.

A *trailing stop loss* (TSL) order is handled in the same way. The only difference is that there is no fixed price attached to a TSL order:

```
In [30]: order = api.create_order('EUR_USD', units=25000,
                                  tsl_distance=0.005,  ❶
                                  suppress=True, ret=True)

In [31]: print_details(order)
Out[31]: ('2020-08-13T12:07:23.204', 'EUR_USD', '25000.0', 1.18341, '0.0')

In [32]: tsl_order = api.get_transaction(tid=int(order['id']) + 1)  ❷

In [33]: tsl_order  ❷
Out[33]: {'id': '1637',
          'time': '2020-08-13T12:07:23.204457044Z',
          'userID': 13834683,
          'accountID': '101-004-13834683-001',
          'batchID': '1635',
          'requestID': '78757241564598562',
          'type': 'TRAILING_STOP_LOSS_ORDER',
          'tradeID': '1636',
          'distance': '0.005',
          'timeInForce': 'GTC',
          'triggerCondition': 'DEFAULT',
          'reason': 'ON_FILL'}

In [34]: (tsl_order['time'][:-7], tsl_order['type'],
          order['price'], tsl_order['distance'])  ❸
Out[34]: ('2020-08-13T12:07:23.204', 'TRAILING_STOP_LOSS_ORDER', 1.18341, '0.005')
```

```
In [35]: time.sleep(1)

In [36]: order = api.create_order('EUR_USD', units=-25000,
                                  suppress=True, ret=True)

In [37]: print_details(order)
Out[37]: ('2020-08-13T12:07:24.551', 'EUR_USD', '-25000.0', 1.1833, '-2.3355')

In [38]: time.sleep(1)
```

❶ The TSL distance is defined in currency units.

❷ Selects and shows the TSL order object data.

❸ Selects and shows some relevant details of the two order objects.

Finally, here is a *take profit* (TP) order. This order requires a fixed TP target price. Therefore, the following code uses the execution price from the previous order to define the TP price in relative terms. Beyond this small difference, the handling is again the same as before:

```
In [39]: tp_price = round(order['price'] + 0.01, 4)
         tp_price
Out[39]: 1.1933

In [40]: order = api.create_order('EUR_USD', units=25000,
                                  tp_price=tp_price,  ❶
                                  suppress=True, ret=True)

In [41]: print_details(order)
Out[41]: ('2020-08-13T12:07:25.712', 'EUR_USD', '25000.0', 1.18344, '0.0')

In [42]: tp_order = api.get_transaction(tid=int(order['id']) + 1)  ❷

In [43]: tp_order  ❷
Out[43]: {'id': '1643',
          'time': '2020-08-13T12:07:25.712531725Z',
          'userID': 13834683,
          'accountID': '101-004-13834683-001',
          'batchID': '1641',
          'requestID': '78757241572993078',
          'type': 'TAKE_PROFIT_ORDER',
          'tradeID': '1642',
          'price': 1.1933,
          'timeInForce': 'GTC',
          'triggerCondition': 'DEFAULT',
          'reason': 'ON_FILL'}

In [44]: (tp_order['time'][:-7], tp_order['type'],
          order['price'], tp_order['price'])  ❸
Out[44]: ('2020-08-13T12:07:25.712', 'TAKE_PROFIT_ORDER', 1.18344, 1.1933)
```

```
In [45]: time.sleep(1)

In [46]: order = api.create_order('EUR_USD', units=-25000,
                                  suppress=True, ret=True)

In [47]: print_details(order)
Out[47]: ('2020-08-13T12:07:27.020', 'EUR_USD', '-25000.0', 1.18332, '-2.5478')
```

❶ The TP target price is defined relative to the previous execution price.

❷ Selects and shows the TP order object data.

❸ Selects and shows some relevant details of the two order objects.

The code so far only deals with transaction details of single orders. However, it is also of interest to have an overview of multiple *historical transactions*. To this end, the following method call provides overview data for all the main orders placed in this section, including P&L data:

```
In [48]: api.print_transactions(tid=int(order['id']) - 22)
          1626 | 2020-08-13T12:07:19.434407966Z |   EUR_USD |      25000.0 |      0.0
          1628 | 2020-08-13T12:07:20.586564454Z |   EUR_USD |     -25000.0 |  -4.0342
          1630 | 2020-08-13T12:07:21.740825489Z |   EUR_USD |      25000.0 |      0.0
          1633 | 2020-08-13T12:07:23.059178023Z |   EUR_USD |     -25000.0 |  -2.9725
          1636 | 2020-08-13T12:07:23.204457044Z |   EUR_USD |      25000.0 |      0.0
          1639 | 2020-08-13T12:07:24.551026466Z |   EUR_USD |     -25000.0 |  -2.3355
          1642 | 2020-08-13T12:07:25.712531725Z |   EUR_USD |      25000.0 |      0.0
          1645 | 2020-08-13T12:07:27.020414342Z |   EUR_USD |     -25000.0 |  -2.5478
```

Yet another method call provides a snapshot of the *account details*. The details shown are from an Oanda demo account that has been in use for quite some time for technical testing purposes:

```
In [49]: api.get_account_summary()
Out[49]: {'id': '101-004-13834683-001',
          'alias': 'Primary',
          'currency': 'EUR',
          'balance': '98541.4272',
          'createdByUserID': 13834683,
          'createdTime': '2020-03-19T06:08:14.363139403Z',
          'guaranteedStopLossOrderMode': 'DISABLED',
          'pl': '-1248.5543',
          'resettablePL': '-1248.5543',
          'resettablePLTime': '0',
          'financing': '-210.0185',
          'commission': '0.0',
          'guaranteedExecutionFees': '0.0',
          'marginRate': '0.0333',
          'openTradeCount': 1,
          'openPositionCount': 1,
```

```
 'pendingOrderCount': 0,
 'hedgingEnabled': False,
 'unrealizedPL': '941.9536',
 'NAV': '99483.3808',
 'marginUsed': '380.83',
 'marginAvailable': '99107.2283',
 'positionValue': '3808.3',
 'marginCloseoutUnrealizedPL': '947.9546',
 'marginCloseoutNAV': '99489.3818',
 'marginCloseoutMarginUsed': '380.83',
 'marginCloseoutPercent': '0.00191',
 'marginCloseoutPositionValue': '3808.3',
 'withdrawalLimit': '98541.4272',
 'marginCallMarginUsed': '380.83',
 'marginCallPercent': '0.00383',
 'lastTransactionID': '1646'}
```

This concludes the discussion of the basics of executing orders with Oanda. All elements are now together to support the deployment of a trading bot. The remainder of this chapter trains a trading bot on Oanda data and deploys it in automated fashion.

Trading Bot

Chapter 11 shows in detail how to train a deep Q-learning trading bot and how to backtest it in vectorized and event-based fashion. This section now repeats selected core steps in this regard based on historical data from Oanda. "Oanda Environment" on page 369 provides a Python module that contains the environment class `OandaEnv` to work with Oanda data. It can be used in the same way as the `Finance` class from Chapter 11.

The following Python code instantiates the learning environment object. During this step, the major data-related parameters driving the learning, validation, and testing are fixed. The `OandaEnv` class allows the inclusion of leverage, which is typical for FX and CFD trading. Leverage amplifies the realized returns, thereby increasing the profit potential but also the loss risks:

```
In [50]: import oandaenv as oe

In [51]: symbol = 'EUR_USD'

In [52]: date = '2020-08-11'

In [53]: features = [symbol, 'r', 's', 'm', 'v']

In [54]: %%time
         learn_env = oe.OandaEnv(symbol=symbol,
                                 start=f'{date} 08:00:00',
                                 end=f'{date} 13:00:00',
                                 granularity='S30',  ❶
```

```
                            price='M',  ❷
                            features=features,  ❸
                            window=20,  ❹
                            lags=3,  ❺
                            leverage=20,  ❻
                            min_accuracy=0.4,  ❼
                            min_performance=0.85  ❽
                           )
        CPU times: user 23.1 ms, sys: 2.86 ms, total: 25.9 ms
        Wall time: 26.8 ms

In [55]: np.bincount(learn_env.data['d'])
Out[55]: array([299, 281])

In [56]: learn_env.data.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 580 entries, 2020-08-11 08:10:00 to 2020-08-11 12:59:30
        Data columns (total 6 columns):
         #   Column   Non-Null Count  Dtype
        ---  ------   --------------  -----
         0   EUR_USD  580 non-null    float64
         1   r        580 non-null    float64
         2   s        580 non-null    float64
         3   m        580 non-null    float64
         4   v        580 non-null    float64
         5   d        580 non-null    int64
        dtypes: float64(5), int64(1)
        memory usage: 31.7 KB
```

❶ Sets the granularity for the data to five seconds

❷ Sets the price type to mid prices

❸ Defines the set of features to be used

❹ Defines the window length for rolling statistics

❺ Specifies the number of lags

❻ Fixes the leverage

❼ Sets the required minimum accuracy

❽ Sets the required minimum performance

In a next step, the validation environment is instantiated, relying on the parameters of the learning environment—apart from the time interval, for obvious reasons.

Figure 12-3 shows the closing prices of EUR/USD as used in the learning, validation, and test environments (from left to right):

```
In [57]: valid_env = oe.OandaEnv(symbol=learn_env.symbol,
                                 start=f'{date} 13:00:00',
                                 end=f'{date} 14:00:00',
                                 granularity=learn_env.granularity,
                                 price=learn_env.price,
                                 features=learn_env.features,
                                 window=learn_env.window,
                                 lags=learn_env.lags,
                                 leverage=learn_env.leverage,
                                 min_accuracy=0,
                                 min_performance=0,
                                 mu=learn_env.mu,
                                 std=learn_env.std
                                )

In [58]: valid_env.data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 100 entries, 2020-08-11 13:10:00 to 2020-08-11 13:59:30
         Data columns (total 6 columns):
          #   Column   Non-Null Count  Dtype
         ---  ------   --------------  -----
          0   EUR_USD  100 non-null    float64
          1   r        100 non-null    float64
          2   s        100 non-null    float64
          3   m        100 non-null    float64
          4   v        100 non-null    float64
          5   d        100 non-null    int64
         dtypes: float64(5), int64(1)
         memory usage: 5.5 KB

In [59]: test_env = oe.OandaEnv(symbol=learn_env.symbol,
                                start=f'{date} 14:00:00',
                                end=f'{date} 17:00:00',
                                granularity=learn_env.granularity,
                                price=learn_env.price,
                                features=learn_env.features,
                                window=learn_env.window,
                                lags=learn_env.lags,
                                leverage=learn_env.leverage,
                                min_accuracy=0,
                                min_performance=0,
                                mu=learn_env.mu,
                                std=learn_env.std
                               )

In [60]: test_env.data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 340 entries, 2020-08-11 14:10:00 to 2020-08-11 16:59:30
         Data columns (total 6 columns):
```

```
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   EUR_USD  340 non-null    float64
 1   r        340 non-null    float64
 2   s        340 non-null    float64
 3   m        340 non-null    float64
 4   v        340 non-null    float64
 5   d        340 non-null    int64
dtypes: float64(5), int64(1)
memory usage: 18.6 KB
```

```
In [61]: ax = learn_env.data[learn_env.symbol].plot(figsize=(10, 6))
         plt.axvline(learn_env.data.index[-1], ls='--')
         valid_env.data[learn_env.symbol].plot(ax=ax, style='-.')
         plt.axvline(valid_env.data.index[-1], ls='--')
         test_env.data[learn_env.symbol].plot(ax=ax, style='-.');
```

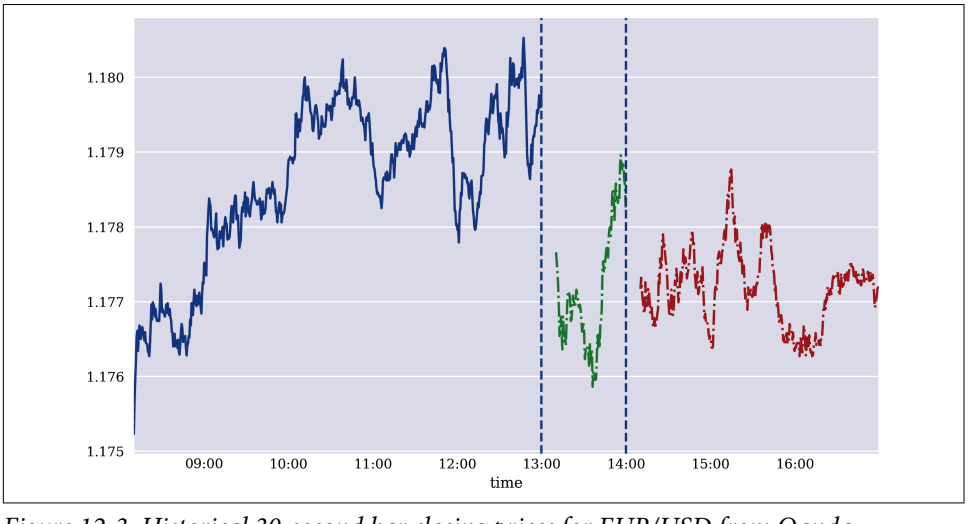

Figure 12-3. Historical 30-second bar closing prices for EUR/USD from Oanda (learning = left, validation = middle, testing = right)

Based on the Oanda environment, the trading bot from Chapter 11 can be trained and validated. The following Python code performs this task and visualizes the performance results (see Figure 12-4):

```
In [62]: import sys
         sys.path.append('../ch11/')  ❶

In [63]: import tradingbot  ❶
         Using TensorFlow backend.

In [64]: tradingbot.set_seeds(100)
         agent = tradingbot.TradingBot(24, 0.001, learn_env=learn_env,
                                       valid_env=valid_env)  ❷
```

```
In [65]: episodes = 31

In [66]: %time agent.learn(episodes)  ❷
         ========================================================================
         episode:  5/31 | VALIDATION | treward:   97 | perf: 1.004 | eps: 0.96
         ========================================================================
         ========================================================================
         episode: 10/31 | VALIDATION | treward:   97 | perf: 1.005 | eps: 0.91
         ========================================================================
         ========================================================================
         episode: 15/31 | VALIDATION | treward:   97 | perf: 0.986 | eps: 0.87
         ========================================================================
         ========================================================================
         episode: 20/31 | VALIDATION | treward:   97 | perf: 1.012 | eps: 0.83
         ========================================================================
         ========================================================================
         episode: 25/31 | VALIDATION | treward:   97 | perf: 0.995 | eps: 0.79
         ========================================================================
         ========================================================================
         episode: 30/31 | VALIDATION | treward:   97 | perf: 0.972 | eps: 0.75
         ========================================================================
         episode: 31/31 | treward:   16 | perf: 0.981 | av: 376.0 | max:  577
         CPU times: user 22.1 s, sys: 1.17 s, total: 23.3 s
         Wall time: 20.1 s

In [67]: tradingbot.plot_performance(agent)  ❸
```

❶ Imports the `tradingbot` module from Chapter 11

❷ Trains and validates the trading bot based on Oanda data

❸ Visualizes the performance results

As discussed in the previous two chapters, the training and validation performances are just an indicator of the trading bot performance.

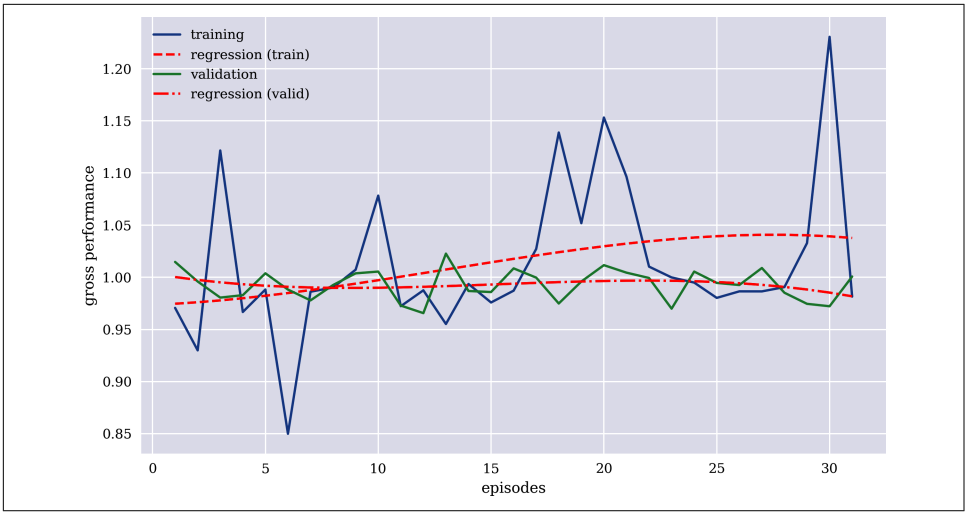

Figure 12-4. Training and validation performance results of the trading bot for Oanda data

The following code implements a vectorized backtest of the trading bot performance for the test environment—again with the same parameters as the learning environment apart from the time interval used. The code makes use of the function `backtest()` as provided in the Python module presented in "Vectorized Backtesting" on page 372. The reported performance numbers include a leverage of 20. This holds true for both the gross performance of the passive benchmark investment and the trading bot over time, as shown in Figure 12-5:

```
In [68]: import backtest as bt

In [69]: env = test_env

In [70]: bt.backtest(agent, env)

In [71]: env.data['p'].iloc[env.lags:].value_counts()  ❶
Out[71]:  1    263
         -1     74
         Name: p, dtype: int64

In [72]: sum(env.data['p'].iloc[env.lags:].diff() != 0)  ❷
Out[72]: 25

In [73]: (env.data[['r', 's']].iloc[env.lags:] * env.leverage).sum(
                ).apply(np.exp)  ❸
Out[73]: r   0.99966
         s   1.05910
         dtype: float64
```

```
In [74]: (env.data[['r', 's']].iloc[env.lags:] * env.leverage).sum(
                ).apply(np.exp) - 1  ❹
Out[74]: r   -0.00034
         s    0.05910
         dtype: float64

In [75]: (env.data[['r', 's']].iloc[env.lags:] * env.leverage).cumsum(
                ).apply(np.exp).plot(figsize=(10, 6));  ❺
```

❶ Shows the total number of long and short positions

❷ Shows the number of trades required to implement the strategy

❸ Calculates the gross performance including leverage

❹ Calculates the net performance including leverage

❺ Visualizes the gross performance over time including leverage

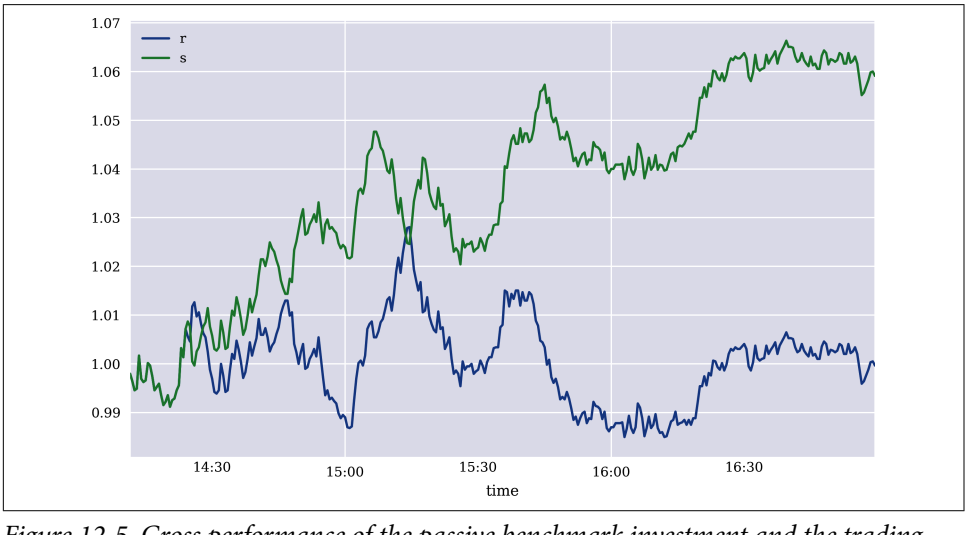

Figure 12-5. Gross performance of the passive benchmark investment and the trading bot over time (including leverage)

Simplified Backtesting

The training and backtesting of the trading bot in this section happen under assumptions that are not realistic. The trading strategy based on the 30-second bars might lead to a large number of trades over a short period of time. Assuming typical transaction costs (bid-ask spreads), such a strategy often is not economically viable. Longer bars or a strategy with fewer trades would be more realistic. However, to allow for a "quick" deployment demo in the next section, the training and backtest are implemented intentionally on the relatively short 30-second bars.

Deployment

This section combines the major elements of the previous sections to deploy the trained trading bot in automated fashion. This is comparable to the point in time at which an AV is prepared to be deployed on the streets. The class `OandaTradingBot` presented in the following code inherits from the `tpqoa` class and adds some helper functions and the trading logic:

```
In [76]: import tpqoa

In [77]: class OandaTradingBot(tpqoa.tpqoa):
             def __init__(self, config_file, agent, granularity, units,
                          verbose=True):
                 super(OandaTradingBot, self).__init__(config_file)
                 self.agent = agent
                 self.symbol = self.agent.learn_env.symbol
                 self.env = agent.learn_env
                 self.window = self.env.window
                 if granularity is None:
                     self.granularity = agent.learn_env.granularity
                 else:
                     self.granularity = granularity
                 self.units = units
                 self.trades = 0
                 self.position = 0
                 self.tick_data = pd.DataFrame()
                 self.min_length = (self.agent.learn_env.window +
                                    self.agent.learn_env.lags)
                 self.pl = list()
                 self.verbose = verbose
             def _prepare_data(self):
                 self.data['r'] = np.log(self.data / self.data.shift(1))
                 self.data.dropna(inplace=True)
                 self.data['s'] = self.data[self.symbol].rolling(
                                                      self.window).mean()
                 self.data['m'] = self.data['r'].rolling(self.window).mean()
                 self.data['v'] = self.data['r'].rolling(self.window).std()
                 self.data.dropna(inplace=True)
```

```
        # self.data_ = (self.data - self.env.mu) / self.env.std  ❶
        self.data_ = (self.data - self.data.mean()) / self.data.std()  ❶
    def _resample_data(self):
        self.data = self.tick_data.resample(self.granularity,
                        label='right').last().ffill().iloc[:-1]  ❷
        self.data = pd.DataFrame(self.data['mid'])  ❷
        self.data.columns = [self.symbol,]  ❷
        self.data.index = self.data.index.tz_localize(None)  ❷
    def _get_state(self):
        state = self.data_[self.env.features].iloc[-self.env.lags:]  ❸
        return np.reshape(state.values, [1, self.env.lags,
                                         self.env.n_features])  ❸
    def report_trade(self, time, side, order):
        self.trades += 1
        pl = float(order['pl'])  ❹
        self.pl.append(pl)  ❹
        cpl = sum(self.pl)  ❺
        print('\n' + 75 * '=')
        print(f'{time} | *** GOING {side} ({self.trades}) ***')
        print(f'{time} | PROFIT/LOSS={pl:.2f} | CUMULATIVE={cpl:.2f}')
        print(75 * '=')
        if self.verbose:
            pprint(order)
            print(75 * '=')
    def on_success(self, time, bid, ask):
        df = pd.DataFrame({'ask': ask, 'bid': bid,
                           'mid': (bid + ask) / 2},
                          index=[pd.Timestamp(time)])
        self.tick_data = self.tick_data.append(df)  ❷
        self._resample_data()  ❷
        if len(self.data) > self.min_length:
            self.min_length += 1
            self._prepare_data()
            state = self._get_state()  ❻
            prediction = np.argmax(
                self.agent.model.predict(state)[0, 0])  ❻
            position = 1 if prediction == 1 else -1  ❻
            if self.position in [0, -1] and position == 1:  ❼
                order = self.create_order(self.symbol,
                        units=(1 - self.position) * self.units,
                                suppress=True, ret=True)
                self.report_trade(time, 'LONG', order)
                self.position = 1
            elif self.position in [0, 1] and position == -1:  ❽
                order = self.create_order(self.symbol,
                        units=-(1 + self.position) * self.units,
                                suppress=True, ret=True)
                self.report_trade(time, 'SHORT', order)
                self.position = -1
```

❶ For demonstration, the normalization is done with the real-time data statistics.[1]

❷ Collects the tick data and resamples it to the required granularity.

❸ Returns the current state of the financial market.

❹ Collects the P&L figures for every trade.

❺ Calculates the cumulative P&L for all trades.

❻ Predicts the market direction and derives the signal (position).

❼ Checks whether the conditions for a *long position* (buy order) are met.

❽ Checks whether the conditions for a *short position* (sell order) are met.

The application of this class is straightforward. First, an object is instantiated, providing as the major input the trained trading bot `agent` from the previous section. Second, the streaming for the instrument to be traded needs to be started. Whenever new tick data arrives, the `.on_success()` method is called, which contains the main logic for both the processing of the tick data and the placement of trades. To speed things up a bit, the deployment example relies, as did the backtesting before, on 30-second bars. In a production context, when managing real money, a longer time interval might be the better choice—if only to reduce the number of trades and therewith the transaction costs:

```
In [78]: otb = OandaTradingBot('../aiif.cfg', agent, '30s',
                               25000, verbose=False)  ❶

In [79]: otb.tick_data.info()
         <class 'pandas.core.frame.DataFrame'>
         Index: 0 entries
         Empty DataFrame
In [80]: otb.stream_data(agent.learn_env.symbol, stop=1000)  ❷

         ========================================================================
         2020-08-13T12:19:32.320291893Z | *** GOING SHORT (1) ***
         2020-08-13T12:19:32.320291893Z | PROFIT/LOSS=0.00 | CUMULATIVE=0.00
         ========================================================================

         ========================================================================
         2020-08-13T12:20:00.083985447Z | *** GOING LONG (2) ***
         2020-08-13T12:20:00.083985447Z | PROFIT/LOSS=-6.80 | CUMULATIVE=-6.80
```

1 This little trick leads more quickly to trades in this particular context given the data used. For real deployment, the statistics from the learning environment data are to be used for the normalization.

```
        ============================================================================

        ============================================================================
        2020-08-13T12:25:00.099901587Z | *** GOING SHORT (3) ***
        2020-08-13T12:25:00.099901587Z | PROFIT/LOSS=-7.86 | CUMULATIVE=-14.66
        ============================================================================

In [81]: print('\n' + 75 * '=')
         print('*** CLOSING OUT ***')
         order = otb.create_order(otb.symbol,
                                  units=-otb.position * otb.units,
                                  suppress=True, ret=True)  ❸
         otb.report_trade(otb.time, 'NEUTRAL', order)  ❸
         if otb.verbose:
             pprint(order)
         print(75 * '=')

         ============================================================================
         *** CLOSING OUT ***

         ============================================================================
         2020-08-13T12:25:16.870357562Z | *** GOING NEUTRAL (4) ***
         2020-08-13T12:25:16.870357562Z | PROFIT/LOSS=-3.19 | CUMULATIVE=-17.84
         ============================================================================
         ============================================================================
```

❶ Instantiates the `OandaTradingBot` object

❷ Starts the streaming of the real-time data and the trading

❸ Closes the final position after a certain number of ticks retrieved

During the deployment, P&L figures are collected in the `pl` attribute, which is a `list` object. Once the trading has stopped, the P&L figures can be analyzed:

```
In [82]: pl = np.array(otb.pl)  ❶

In [83]: pl  ❶
Out[83]: array([ 0.    , -6.7959, -7.8594, -3.1862])

In [84]: pl.cumsum()  ❷
Out[84]: array([  0.    ,  -6.7959, -14.6553, -17.8415])
```

❶ P&L figures for all trades

❷ Cumulative P&L figures

The simple deployment example illustrates that one can trade algorithmically and in automated fashion with a deep Q-learning trading bot in less than 100 lines of Python code. The major prerequisite is the trained trading bot (i.e., an instance of the

`tradingbot` class). Many important aspects are intentionally left out here. For example, in a production environment, one would probably like to persist the data. One would also like to persist the order objects. Measures to make sure that the socket connection is still alive are also important (for example, by monitoring a heartbeat). Overall, security, reliability, logging, and monitoring are not really addressed. Some more details in this regard are provided in Hilpisch (2020).

The Python script in "Oanda Trading Bot" on page 373 presents a standalone executable version of the `OandaTradingBot` class. This represents a major step toward a more robust deployment option as compared to an interactive context such as Jupyter Notebook or Jupyter Lab. The script also includes functionality to add SL, TSL, or TP orders for the execution. The script expects a pickled version of the `agent` object in the current working directory. The following Python code pickles the object for later usage by the script:

```
In [85]: import pickle

In [86]: pickle.dump(agent, open('trading.bot', 'wb'))
```

Conclusions

This chapter discusses central aspects of the execution of an algorithmic trading strategy and the deployment of a trading bot. The Oanda trading platform provides directly or indirectly with its v20 API all necessary capabilities to do the following:

- Retrieve historical data
- Train and backtest a trading bot (deep Q-learning agent)
- Stream real-time data
- Place market (and limit) orders
- Make use of SL, TSL, and TP orders
- Deploy a trading bot in an automated manner

The prerequisites to implement all these steps are a demo account with Oanda, standard hardware and software (open source only), and a stable internet connection. In other words, the barriers of entry to algorithmic trading for the purposes of exploiting economic inefficiencies are pretty low. This is in stark contrast, for example, to the training, design, and construction of AVs for deployment on public streets—the budgets of companies in the AV space run into the billions of dollars. In other words, the finance domain has distinctive advantages compared to other industries and domains with regard to the real-world deployment of AI agents, such as trading bots, as focused on in this and the previous chapter.

References

Books and papers cited in this chapter:

Hilpisch, Yves. 2020. *Python for Algorithmic Trading: From Idea to Cloud Deployment.* Sebastopol: O'Reilly.

Litman, Todd. 2020. "Autonomous Vehicle Implementation Predictions." *Victoria Transport Policy Institute*. https://oreil.ly/ds7YM.

Python Code

This section contains code used and referenced in the main body of the chapter.

Oanda Environment

The following is the Python module with the `OandaEnv` class to train a trading bot based on historical Oanda data:

```
#
# Finance Environment
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
#
import math
import tpqoa
import random
import numpy as np
import pandas as pd


class observation_space:
    def __init__(self, n):
        self.shape = (n,)


class action_space:
    def __init__(self, n):
        self.n = n

    def sample(self):
        return random.randint(0, self.n - 1)


class OandaEnv:
    def __init__(self, symbol, start, end, granularity, price,
                 features, window, lags, leverage=1,
                 min_accuracy=0.5, min_performance=0.85,
```

```
                mu=None, std=None):
        self.symbol = symbol
        self.start = start
        self.end = end
        self.granularity = granularity
        self.price = price
        self.api = tpqoa.tpqoa('../aiif.cfg')
        self.features = features
        self.n_features = len(features)
        self.window = window
        self.lags = lags
        self.leverage = leverage
        self.min_accuracy = min_accuracy
        self.min_performance = min_performance
        self.mu = mu
        self.std = std
        self.observation_space = observation_space(self.lags)
        self.action_space = action_space(2)
        self._get_data()
        self._prepare_data()

    def _get_data(self):
        ''' Method to retrieve data from Oanda.
        '''
        self.fn = f'../../source/oanda/'  ❶
        self.fn += f'oanda_{self.symbol}_{self.start}_{self.end}_'  ❷
        self.fn += f'{self.granularity}_{self.price}.csv'  ❷
        self.fn = self.fn.replace(' ', '_').replace('-', '_').replace(':', '_')
        try:
            self.raw = pd.read_csv(self.fn, index_col=0, parse_dates=True)  ❸
        except:
            self.raw = self.api.get_history(self.symbol, self.start,
                                            self.end, self.granularity,
                                            self.price)  ❹
            self.raw.to_csv(self.fn)  ❺
        self.data = pd.DataFrame(self.raw['c'])  ❻
        self.data.columns = [self.symbol]  ❼

    def _prepare_data(self):
        ''' Method to prepare additional time series data
            (such as features data).
        '''
        self.data['r'] = np.log(self.data / self.data.shift(1))
        self.data.dropna(inplace=True)
        self.data['s'] = self.data[self.symbol].rolling(self.window).mean()
        self.data['m'] = self.data['r'].rolling(self.window).mean()
        self.data['v'] = self.data['r'].rolling(self.window).std()
        self.data.dropna(inplace=True)
        if self.mu is None:
            self.mu = self.data.mean()
            self.std = self.data.std()
        self.data_ = (self.data - self.mu) / self.std
```

```
        self.data['d'] = np.where(self.data['r'] > 0, 1, 0)
        self.data['d'] = self.data['d'].astype(int)

    def _get_state(self):
        ''' Privat method that returns the state of the environment.
        '''
        return self.data_[self.features].iloc[self.bar -
                                    self.lags:self.bar].values

    def get_state(self, bar):
        ''' Method that returns the state of the environment.
        '''
        return self.data_[self.features].iloc[bar - self.lags:bar].values

    def reset(self):
        ''' Method to reset the environment.
        '''
        self.treward = 0
        self.accuracy = 0
        self.performance = 1
        self.bar = self.lags
        state = self._get_state()
        return state

    def step(self, action):
        ''' Method to step the environment forwards.
        '''
        correct = action == self.data['d'].iloc[self.bar]
        ret = self.data['r'].iloc[self.bar] * self.leverage
        reward_1 = 1 if correct else 0  ❽
        reward_2 = abs(ret) if correct else -abs(ret)  ❾
        reward = reward_1 + reward_2 * self.leverage  ❿
        self.treward += reward_1
        self.bar += 1
        self.accuracy = self.treward / (self.bar - self.lags)
        self.performance *= math.exp(reward_2)
        if self.bar >= len(self.data):
            done = True
        elif reward_1 == 1:
            done = False
        elif (self.accuracy < self.min_accuracy and
              self.bar > self.lags + 15):
            done = True
        elif (self.performance < self.min_performance and
              self.bar > self.lags + 15):
            done = True
        else:
            done = False
        state = self._get_state()
        info = {}
        return state, reward, done, info
```

❶ Defines the path for the data file

❷ Defines the filename of the data file

❸ Reads the data if a corresponding data file exists

❹ Retrieves the data for the API if no such file exists

❺ Writes the data as a `CSV` file to disk

❻ Selects the column with the closing prices

❼ Renames the column to the instrument name (symbol)

❽ Reward for correct prediction

❾ Reward for the realized performance (return)

❿ Combined reward for prediction and performance

Vectorized Backtesting

The following is the Python module with the helper function `backtest` to generate the data to do a vectorized backtest for a deep Q-learning trading bot. The code is also used in Chapter 11:

```
#
# Vectorized Backtesting of
# Trading Bot (Financial Q-Learning Agent)
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import numpy as np
import pandas as pd
pd.set_option('mode.chained_assignment', None)

def reshape(s, env):
    return np.reshape(s, [1, env.lags, env.n_features])

def backtest(agent, env):
    done = False
    env.data['p'] = 0
    state = env.reset()
    while not done:
        action = np.argmax(
            agent.model.predict(reshape(state, env))[0, 0])
        position = 1 if action == 1 else -1
```

```
        env.data.loc[:, 'p'].iloc[env.bar] = position
        state, reward, done, info = env.step(action)
    env.data['s'] = env.data['p'] * env.data['r']
```

Oanda Trading Bot

The following is the Python script with the `OandaTradingBot` class and code to deploy the class:

```
#
# Oanda Trading Bot
# and Deployment Code
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import sys
import tpqoa
import pickle
import numpy as np
import pandas as pd

sys.path.append('../ch11/')


class OandaTradingBot(tpqoa.tpqoa):
    def __init__(self, config_file, agent, granularity, units,
                 sl_distance=None, tsl_distance=None, tp_price=None,
                 verbose=True):
        super(OandaTradingBot, self).__init__(config_file)
        self.agent = agent
        self.symbol = self.agent.learn_env.symbol
        self.env = agent.learn_env
        self.window = self.env.window
        if granularity is None:
            self.granularity = agent.learn_env.granularity
        else:
            self.granularity = granularity
        self.units = units
        self.sl_distance = sl_distance
        self.tsl_distance = tsl_distance
        self.tp_price = tp_price
        self.trades = 0
        self.position = 0
        self.tick_data = pd.DataFrame()
        self.min_length = (self.agent.learn_env.window +
                           self.agent.learn_env.lags)
        self.pl = list()
        self.verbose = verbose
    def _prepare_data(self):
        ''' Prepares the (lagged) features data.
        '''
```

```
        self.data['r'] = np.log(self.data / self.data.shift(1))
        self.data.dropna(inplace=True)
        self.data['s'] = self.data[self.symbol].rolling(self.window).mean()
        self.data['m'] = self.data['r'].rolling(self.window).mean()
        self.data['v'] = self.data['r'].rolling(self.window).std()
        self.data.dropna(inplace=True)
        self.data_ = (self.data - self.env.mu) / self.env.std
    def _resample_data(self):
        ''' Resamples the data to the trading bar length.
        '''
        self.data = self.tick_data.resample(self.granularity,
                                label='right').last().ffill().iloc[:-1]
        self.data = pd.DataFrame(self.data['mid'])
        self.data.columns = [self.symbol,]
        self.data.index = self.data.index.tz_localize(None)
    def _get_state(self):
        ''' Returns the (current) state of the financial market.
        '''
        state = self.data_[self.env.features].iloc[-self.env.lags:]
        return np.reshape(state.values, [1, self.env.lags, self.env.n_features])
    def report_trade(self, time, side, order):
        ''' Reports trades and order details.
        '''
        self.trades += 1
        pl = float(order['pl'])
        self.pl.append(pl)
        cpl = sum(self.pl)
        print('\n' + 71 * '=')
        print(f'{time} | *** GOING {side} ({self.trades}) ***')
        print(f'{time} | PROFIT/LOSS={pl:.2f} | CUMULATIVE={cpl:.2f}')
        print(71 * '=')
        if self.verbose:
            pprint(order)
            print(71 * '=')
    def on_success(self, time, bid, ask):
        ''' Contains the main trading logic.
        '''
        df = pd.DataFrame({'ask': ask, 'bid': bid, 'mid': (bid + ask) / 2},
                          index=[pd.Timestamp(time)])
        self.tick_data = self.tick_data.append(df)
        self._resample_data()
        if len(self.data) > self.min_length:
            self.min_length += 1
            self._prepare_data()
            state = self._get_state()
            prediction = np.argmax(self.agent.model.predict(state)[0, 0])
            position = 1 if prediction == 1 else -1
            if self.position in [0, -1] and position == 1:
                order = self.create_order(self.symbol,
                        units=(1 - self.position) * self.units,
                        sl_distance=self.sl_distance,
                        tsl_distance=self.tsl_distance,
```

```
                        tp_price=self.tp_price,
                        suppress=True, ret=True)
                self.report_trade(time, 'LONG', order)
                self.position = 1
            elif self.position in [0, 1] and position == -1:
                order = self.create_order(self.symbol,
                        units=-(1 + self.position) * self.units,
                        sl_distance=self.sl_distance,
                        tsl_distance=self.tsl_distance,
                        tp_price=self.tp_price,
                        suppress=True, ret=True)
                self.report_trade(time, 'SHORT', order)
                self.position = -1


if __name__ == '__main__':
    agent = pickle.load(open('trading.bot', 'rb'))
    otb = OandaTradingBot('../aiif.cfg', agent, '5s',
                          25000, verbose=False)
    otb.stream_data(agent.learn_env.symbol, stop=1000)
    print('\n' + 71 * '=')
    print('*** CLOSING OUT ***')
    order = otb.create_order(otb.symbol,
                    units=-otb.position * otb.units,
                    suppress=True, ret=True)
    otb.report_trade(otb.time, 'NEUTRAL', order)
    if otb.verbose:
        pprint(order)
    print(71 * '=')
```