REINFORCEMENT LEARNING FOR AUTOMATED TRADING

Reid Falconer[a], Sam MacIntyre[a], Hector Cano[a], Maximilian Zebhauser[a]

[a] *Barcelona Graduate School of Economics, Barcelona, Spain*

Keywords: Reinforcement Learning, Deep Q-Learning, Automated Trading, Neural Networks

1. Introduction

Algorithmic trading for stocks is attractive for both researchers and market practitioners. Existing approaches for algorithmic trading can be categorised into knowledge-based methods and machine learning (ML) based methods. Knowledge-based methods design trading strategies based on either financial research or trading experience; ML-based methods, in contrast, learn trading strategies from the historical market data. A distinct advantage of the ML-based methods is that they can discover profitable patterns that are not yet known to people.

Among various ML methods, reinforcement learning (RL) is particularly exciting and is considered a third ML paradigm alongside unsupervised and supervised learning. Nevertheless, unlike the other approaches, RL considers the whole problem of a goal-directed agent that interacts in an uncertain environment. This approach involves learning what actions are necessary to take in order to maximise a numerical reward signal.

The most important distinguishing features of a RL problem are:

1. They behave as closed-loop problems given that its learned actions influence later inputs

Email addresses: reid.falconer@barcelonagse.eu (Reid Falconer), sam.macintyre@barcelonagse.eu (Sam MacIntyre), hector.cano@barcelonagse.eu (Hector Cano), maximilian.zebhauser@barcelonagse.eu (Maximilian Zebhauser)

2. Learners must try different operations to discover which strategy yields the most reward

3. Actions may affect next situations and all subsequent rewards (Sutton et al., 1998)

Among its main applications are: resources management in computer clusters, games, traffic light control and robotics Mnih et al. (2013). This project aims to apply RL techniques to make decisions in the stock market given that it involves the interaction of an active agent that has to make decisions based on an imperfect information environment while also interacting with other market participants. Some previous findings indicate that RL can be successfully applied to the portfolio problem and its performance exceeds the supervised learning approach (Neuneier, 1996) and Q-learning algorithm operates better than kernel-based methods (Bertoluzzo and Corazza, 2012).

In this paper, we apply the deep Q-learning approach to algorithmic trading. Our goal is to build a deep Q-learning system that determines when to buy, sell or hold based on the current and historical market data. Our experiments on the both Apple (AAPL) and Wawel (WWL) stocks demonstrate that the deep Q-learning system is highly effective and that the deep Q-learning model outperforms the benchmarks such as a random decision policy and a buy and hold strategy.

The paper is organised as follows: Firstly, a conceptual framework is presented, highlighting the underlying principles of reinforcement learning. Section 3 describes the Q-learning approach, and Section 4 presents the implementation details of the deep Q-learning system. Section 5 presents the data, settings and results. Finally, Section 6 concludes.

2. Reinforcement Learning

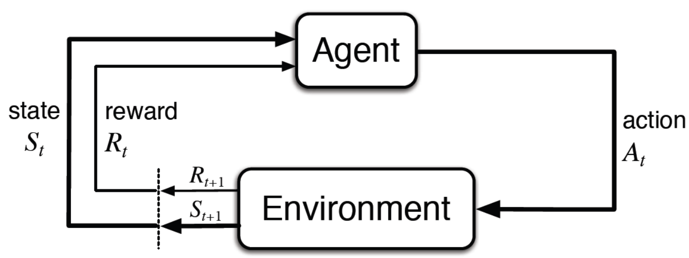
Before delving into the specifics of employing reinforcement learning to the problem of automated trading, it will be informative to discuss the general theory and its underlying principles.

Reinforcement learning aims to maximise a given reward signal by undertaking certain actions (in a restricted space). In this framework, an agent must take the state of the environment as input and take actions to alter the future state. A measurable goal related to the environment is also necessary for the problem formulation. Beyond this, each reinforcement learning problem contains four sub-elements: a *policy*, a *reward signal* and a *value function* (Sutton et al., 1998).

The policy defines the agent's actions in different environment states. The reward signal defines the goal and should be maximised throughout the learning process. The value function maps the current state to a value so the agent can make optimal longer run decisions. It can be seen as the expected total future reward that can be obtained beginning from that state. Most of the challenges associated with the implementation of reinforcement learning derive from the estimation of the value function.

Formally, we construct a Markov Decision Process (MDP). In an ideal situation, we would have access to the value function directly in tabular form when we have a tractable action and state space.

Figure 1: Agent and Environment Interaction



Source: Sutton et al. (1998)

At a sequence of discrete time steps $t = 0, 1, 2, 3...$, the agent interacts with the environment. At each step t, the agent receives state information $S_t \in \mathcal{S}$ and performs an action $A_t \in \mathcal{A}(s)$. As a consequence of the action, the agent receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and transitions to a new state S_{t+1}.

In the context of a *Markov* decision process, the future rewards (R_t) and states (S_t) only depend on the previous state and action.

The general reinforcement learning paradigm involves finding an optimal policy π to maximise the expected discounted return. The discount factor is required to ensure that rewards in the distant future are less valuable than current rewards.

$$G_t \doteq R_{t+1} + \gamma R_t + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Value and action-value functions allow the actions of the agent to be assessed under the implementation of a particular policy. The value function and action-value functions respectively are defined below:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right], \text{ for all } s \in \mathcal{S}$$
$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right]$$

Ideally, the value function is decomposed into the following (known as the *Bellman's Equation*):

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p\left(s', r | s, a\right) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s', r} p\left(s', r | s, a\right) \left[r + \gamma v_\pi\left(s'\right)\right], \quad \text{for all } s \in S \end{aligned}$$

Both expressions relate to a specific state and action taken at any time t.

A reinforcement learning problem principally involves pursuing the optimal policy π which is said to maximise the value and action-value functions:

$$v_*(s) \doteq \max_\pi v_\pi(s)$$
$$q_*(s, a) \doteq \max_\pi q_\pi(s, a)$$

In the most simple cases where the value and action-value functions are specified, dynamic programming can be used to derive the optimal policy π.

In the algorithmic trading problem, the value function cannot be ascertained easily in this way. To deal with these situations and arbitrarily large state space, approximate solution methods must be used. This is known as a partially observable Markov decision process as the state is only observed indirectly and cannot be fully known (we cannot know the trading behaviour of other agents for example) and we do not have access to the transition probabilities between states.

Q-Learning is a technique whereby the value functions are repeatedly estimated based on the rewards of our actions and assumes no prior model specification.

3. Q-Learning

Q-learning attempts to estimate q_* (optimal action-value function) without any regard for the policy followed. From a high-level perspective, the Q-Learning algorithm proceeds by randomly initialising Q, perform actions, measure reward and update Q accordingly (see Figure 2). The final output after a training period should be a stable approximation of the q_*.

Figure 2: Q-Learning Algorithm

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until S is terminal

Source: Sutton et al. (1998)

Notice the Bellman equation appearing in the update phase of the algorithm.

4. Deep Q Learning

An extension of this idea (which we aim to employ) is to use neural networks to approximate the Q-function ($Q(s, a)$). In situations where the state space is enumerable, we can produce a Q table which specifies the action-value function for each possible state. However, if the state space is intractable or very large, this is generally not possible or computationally intensive. A neural network is an appropriate tool for our use case due to the infinite nature of the state space. Estimating a table corresponding to every possible state would be excessive in regards to memory requirements.

To train the neural network on the state space, we must define a loss function. The Bellman Equation defines the optimal result; thus we can use this to calculate our loss as follows:

$$\hat{Q}(s,a) = R(s,a) + \gamma \max_{a' \in A} Q(s,a)$$
$$\text{Loss} = \|Q - \hat{Q}\|_2$$

In general, a partition of the data is used to train the neural network and approximate the q_* function, and this is then used as our action-value function for deciding the optimal policy.

A neural network with two hidden layers is implemented using Tensorflow[1] in our case.

4.1. Problem Formulation - Algorithmic Trading

Now we must formulate the trading problem as a Markov Decision Process and define the states, actions and rewards (Xiong et al., 2018).

- State: $\mathcal{S} = \{prices, holdings, balance\}$ where *prices* refers to the current prices of all the stocks in our portfolio, *holdings* the quantity of each stock held and *balance* as the total portfolio value. In our simple variant, we are only trading one stock but include a history of prices also (default 200)

- Actions: $\mathcal{A} = \{buy, sell, hold\}$ For simplicity and computational tractability, we restrict our action space to buy 1 stock, sell 1 stock or hold.

[1]TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks.

- Rewards: $R_t \in \mathcal{R}$ can be defined as the change in the portfolio value due to an action A_t. Our Reward was defined as $R_t = balance_t - balance_{t-1}$
- Policy: π which is governs the trading strategy at state S. We converge on the optimal policy by approximating the q_* function.
- Action-value function: $q_\pi(s, a)$ as defined above. The expected reward we obtain by following policy π, choosing action A while in state S. The action-value function is approximated by a neural network.

No transaction cost is considered in this study, and the algorithmic trader can only trade with a single stock, Apple (AAPL) or Wawel (WWL). Furthermore, a negative portfolio is not permissible ($balance_t \geq 0$ for all t) and the trader can only sell owned stock ($holdings_t \geq 1$). In our primary model, the initial $balance_0$ is set to \$1000 and $price_0$ to the price of the stock at our chosen start time t_0

5. Data, Settings and Results

The proposed deep Q-learning system is evaluated on two stocks: Apple (AAPL) and Wawel (WWL). We first describe the data and configurations of the experiment and then present the performance results.

5.1. Data and Settings

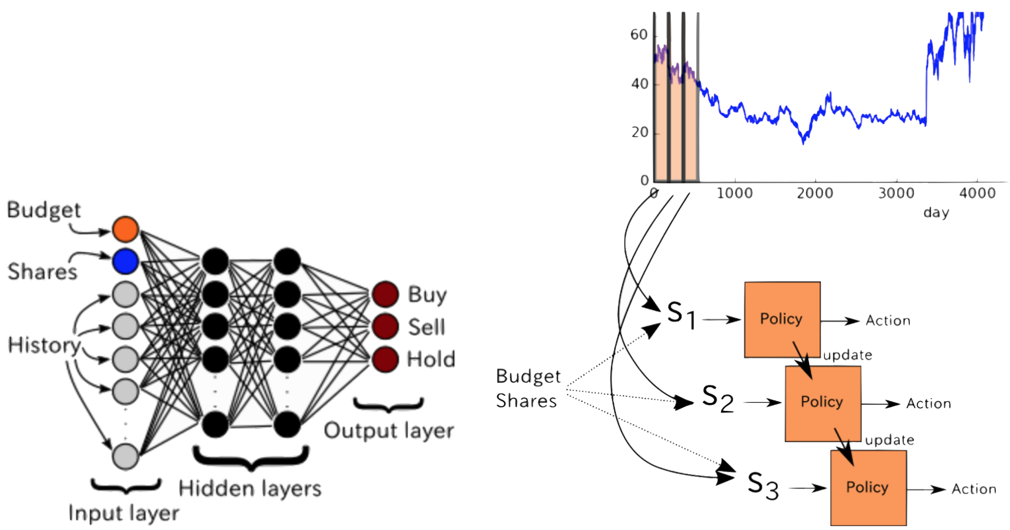
The databases used in the experiment involve nine years of daily data on Apple and Wawel stocks[2], ranging from 2010-01-01 through 2019-03-01. The databases are divided into training data sets (2010-01-01 — 2012-04-13) and test sets (2012-04-16 — 2019-02-28). Only the daily closing price is used in this study, though other features can be easily incorporated into the model.

We compare the deep Q-learning system with two benchmark strategies: a random decisions policy (RAND) and a buy-and-hold (BH) strategy. For deep Q-learning, the training datasets are used to initialise the deep Q-network, and then the system runs in an online

[2]All data was downloaded using Yahoo finance

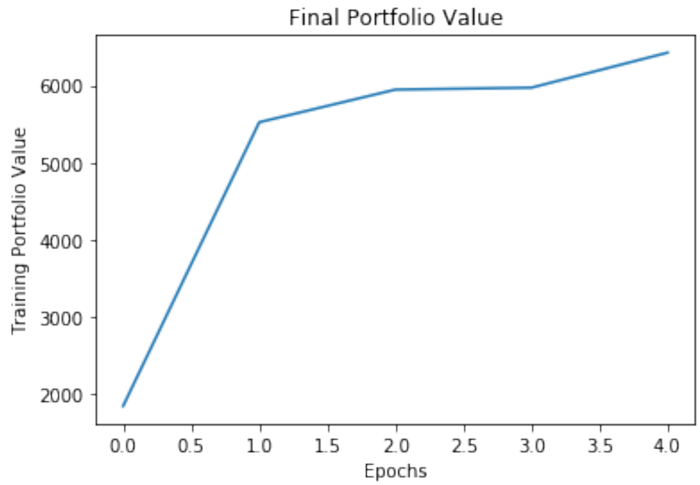
fashion where trading decision making and model adaptation are conducted simultaneously. For the Q-learning part, the discount factor is set to $\gamma = 0.5$. For the deep-learning part, the architecture of the deep Q-network involves four layers in total (two are hidden). The input units (features) are composed by the current budget ($balance_t$), the current number of stocks held ($holdings_t$) and 200 days of stock history while the output units correspond to the three actions in trading (see Figure 3). The learning rate for Q-network is 0.01, and the training stops after 5 epochs. Furthermore, we introduce a new hyper-parameter ϵ (set at $\epsilon = 0.9$) to keep our solution from getting "stuck" when applying the same action over and over. Thus the algorithm exploits the best option with probability ϵ and explores a random option with probability $1 - \epsilon$.

Figure 3: The architecture of the deep Q-network and the rolling window scheme



As mentioned above, to ensure that the Q function is being learned correctly and trending towards an optimal policy, we train the neural network over five epochs. From the plot below (Figure 4), the final portfolio value achieved continues to increase, signalling convergence towards an optimal policy. The hyper-parameters were tuned to derive maximum gain.

Figure 4: Illustrative convergence of the deep Q-network



5.2. Results

The results of the three trading approaches (BH, RAND, and Deep Q-learning) on the AAPL test set are presented in Figure 5, and the results on the WWL test set are shown in Figure 6. In each figure, the green line illustrates the Deep Q-learning decision policy while the red and blue lines depict the BH and RAND strategies respectively. It can be seen that on both of the two test sets, the deep Q-learning system accumulates more value than the other two systems. A more detailed numerical comparison is shown in Table 1 where we report two widely used measures for stock trading: accumulated return and the Sharpe ratio. The Sharpe ratio is defined as:

$$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p}$$

where R_f is the risk-free rate (assumed to be 8%), R_p the return of the portfolio and σ_p is the standard deviation of the portfolio. A larger Sharpe Ratio indicates that the portfolio achieves a better return for its volatility. In a practical setting, this would be an attractive feature for a potential investor. Thus, it can be seen that our deep Q-learning system outperforms the other two methods in terms of total return and the Sharpe Ratio.

Therefore, the results show our deep Q-learning model performs well on both the two stocks (AAPL being a strong stock with a stable upward trend and WWL being a more volatile stock with no apparent trend). This advantage of deep Q-learning can be at-

Table 1: Comparison of Trading Performance

	AAPL			WLL		
	DQL	BH	RAND	DQL	BH	RAND
Accumulated Return(%)	1.67	1.33	0.59	0.64	0.41	0.18
Sharp Ratio	0.29	0.22	0.26	0.25	0.20	0.09

tributed to two advantages. One of them is the ability to detect the status of the market from the raw and noisy data, and the other is the online nature that adapts itself to new market status quickly. More interesting observations can be found in the portfolio action plots (see Figure 7 and Figure 8). From the positions held by the Q-learning system, it seems that it has learned how to take different actions in different market situations which can largely be attributed to the power of the deep Q-network in discovering the status of the market from the noisy historical price signals.

Figure 5: The performance of various trading strategies on AAPL

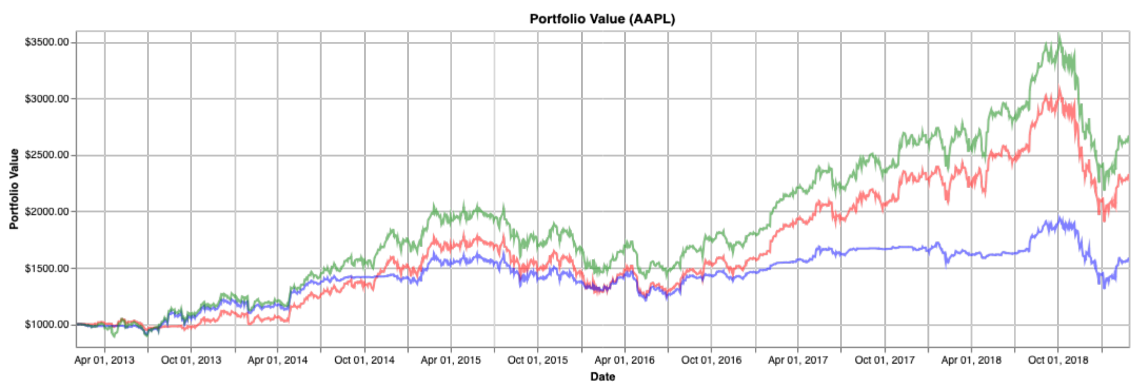


Figure 6: The performance of various trading strategies on WWL

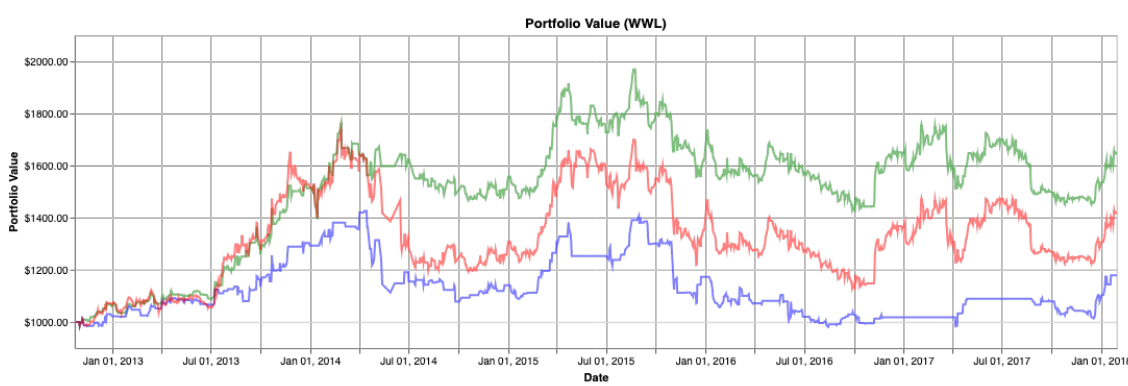


Figure 7: The positions held by various trading strategies on AAPL

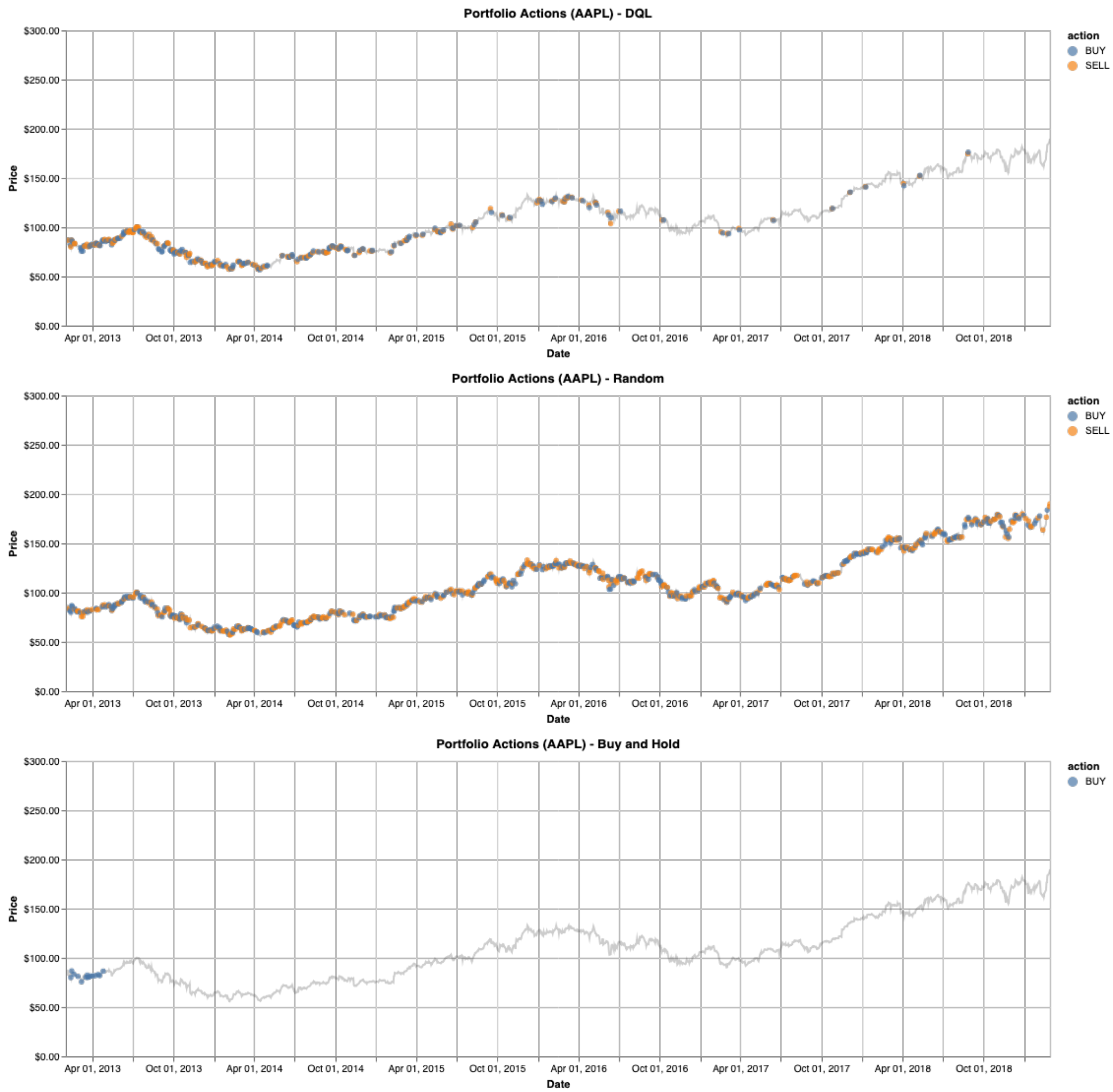
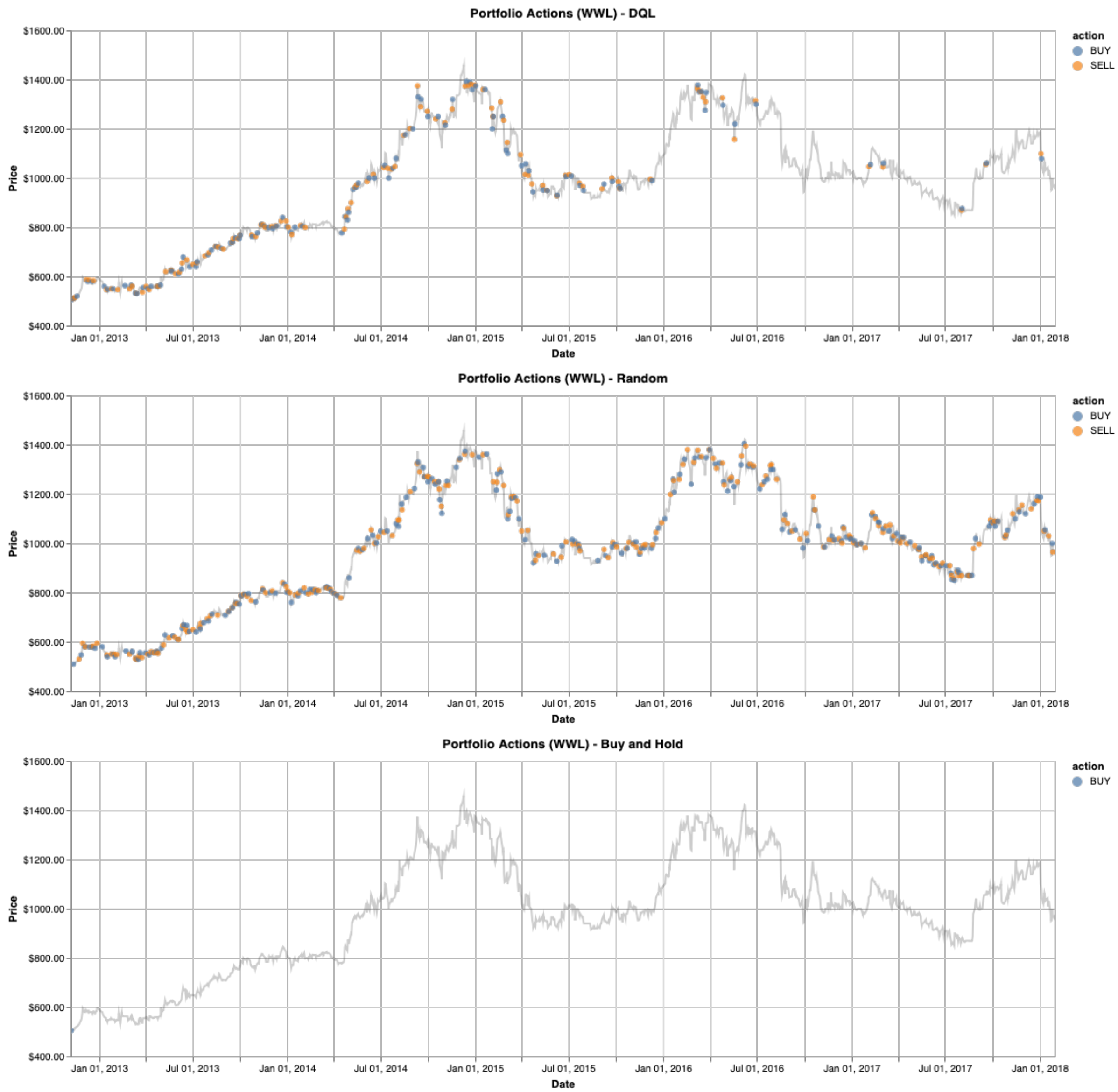


Figure 8: The position held by various trading strategies on WWL



6. Conclusion

The investigation aimed to implement reinforcement learning in the context of stock trading. A deep Q-Learning approach was used to approximate the Q-function (action-value function) and develop a trading policy. Through our experiments with AAPL and WLL stocks, we have managed to show that with minimal training and hyper-parameter tuning, our agent can outperform random actions and a Buy and Hold strategy. We have also demonstrated the agent's ability to manage the volatility of the portfolio effectively.

There are many possible future developments that could be explored such as using multiple stocks, modifying the reward function and doing further hyper-parameter tuning. Furthermore, the initial budget appeared to be a strong determinant of the behaviour of our agent. A more thorough investigation into this and its implications for algorithmic trading would be an interesting extension.

7. References

Bertoluzzo, F. and Corazza, M. (2012), 'Testing different reinforcement learning configurations for financial trading: Introduction and applications', *Procedia Economics and Finance* **3**, 68–77.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013), 'Playing atari with deep reinforcement learning', *arXiv preprint arXiv:1312.5602* .

Neuneier, R. (1996), Optimal asset allocation using adaptive dynamic programming, *in* 'Advances in Neural Information Processing Systems', pp. 952–958.

Sutton, R. S., Barto, A. G. et al. (1998), *Introduction to reinforcement learning*, Vol. 135, MIT press Cambridge.

Xiong, Z., Liu, X.-Y., Zhong, S., Walid, A. et al. (2018), 'Practical deep reinforcement learning approach for stock trading', *arXiv preprint arXiv:1811.07522* .

In [65]:

```
import warnings
warnings.filterwarnings("ignore")
```

In [66]:

```
from matplotlib import pyplot as plt
import numpy as np
import random
import tensorflow as tf
import fix_yahoo_finance as yf
from datetime import datetime
import time
import copy
import pandas as pd
import altair as alt
import seaborn as sns
```

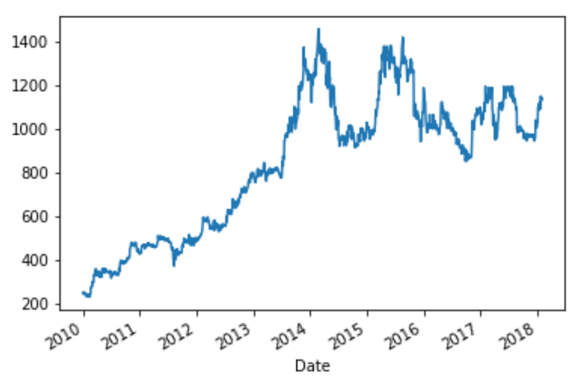
In [67]:

```
def plot_prices(train_prices, test_prices=None):
    plt.xlabel('time')
    plt.ylabel('price')
    plt.plot(range(len(train_prices)), train_prices, 'b')
    if test_prices is not None:
        plt.plot(range(len(train_prices), len(train_prices) + len(test_prices)), test_prices, 'g')
    plt.show()
```

In [68]:

```
data = yf.download('WWL','2010-01-01','2019-03-01')
data.Close.plot()
plt.show()
```

```
[*********************100%***********************]  1 of 1 downloaded
```



In [69]:

```
df = yf.download('WWL','2010-01-01','2019-03-01')
df.index.name = 'Date'
df.reset_index(inplace=True)
# filter out the desired features
df = df[['Date', 'Adj Close']]
# rename feature column names
df = df.rename(columns={'Adj Close': 'actual', 'Date': 'date'})
# convert dates from object to DateTime type
dates = df['date']
dates = pd.to_datetime(dates, infer_datetime_format=True)
df['date'] = dates
```

```
[*********************100%***********************]  1 of 1 downloaded
```

In [70]:

```
class DecisionPolicy:
    # A Given a state, the decision policy will calculate the next action to take
    def select_action(self, current_state):
        pass
    # Improve the Q-function from a new experience of taking an action
    def update_q(self, state, action, reward, next_state):
        pass
```

In [71]:

```
class RandomDecisionPolicy(DecisionPolicy):
    # Inherit from DecisionPolicy to implement its functions
    def __init__(self, actions):
        self.actions = actions
    # Randomly choose the next action
    def select_action(self, current_state, step):
        action = random.choice(self.actions)
        return action
```

In [72]:

```
class QLearningDecisionPolicy(DecisionPolicy):
    # Set the hyper-parameters from the Q-function.
    # To keep the solution from getting "stuck" when applying the same action over and over.
    # The lesser the epsilon value, the more often it will randomly explore new actions.
    def __init__(self, actions, input_dim):
        self.epsilon = 0.95
        self.gamma = 0.5
        self.actions = actions
        output_dim = len(actions) # output dimentions
        h1_dim = 20 # Set the number of hidden nodes in the neural networks

        self.x = tf.placeholder(tf.float32, [None, input_dim]) # input vector
        self.y = tf.placeholder(tf.float32, [output_dim]) # output vector
        W1 = tf.Variable(tf.random_normal([input_dim, h1_dim])) # weights from input to hidden
        b1 = tf.Variable(tf.constant(0.1, shape=[h1_dim])) # biases from input to hidden
        h1 = tf.nn.relu(tf.matmul(self.x, W1) + b1) # hidden layer vector
        W2 = tf.Variable(tf.random_normal([h1_dim, output_dim])) # weights from hidden to output
        b2 = tf.Variable(tf.constant(0.1, shape=[output_dim])) # biases from hidden to output
        self.q = tf.nn.relu(tf.matmul(h1, W2) + b2) # output of neural network

        # Set loss as a squared error and use an optimizer
        loss = tf.square(self.y - self.q)
        self.train_op = tf.train.AdamOptimizer(0.001).minimize(loss)
        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())

    def select_action(self, current_state, step):
        threshold = min(self.epsilon, step / 1000.)
        if random.random() < threshold:
            # Exploit best option with probability epsilon
            action_q_vals = self.sess.run(self.q, feed_dict={self.x: current_state})
            action_idx = np.argmax(action_q_vals)  # TODO: replace w/ tensorflow's argmax
            action = self.actions[action_idx]
        else:
            # Explore random option with probability 1 - epsilon
            action = self.actions[random.randint(0, len(self.actions) - 1)]
        return action

    # Update the Q-function by updating its model parameters
    def update_q(self, state, action, reward, next_state):
        action_q_vals = self.sess.run(self.q, feed_dict={self.x: state})
        next_action_q_vals = self.sess.run(self.q, feed_dict={self.x: next_state})
        next_action_idx = np.argmax(next_action_q_vals)
        current_action_idx = self.actions.index(action)
        action_q_vals[0, current_action_idx] = reward + self.gamma * next_action_q_vals[0,
next_action_idx]
        action_q_vals = np.squeeze(np.asarray(action_q_vals))
        self.sess.run(self.train_op, feed_dict={self.x: state, self.y: action_q_vals})
```

In [73]:

```
def run_simulation(policy, initial_budget, initial_num_stocks, prices, hist, learn=True):
    # Initialize values that depend on computing the net worth of a portfolio
    budget = initial_budget
    num_stocks = initial_num_stocks
    share_value = 0
    transitions = list()
    history = []
    portfolio_history = []
    for i in range(len(prices) - hist - 1):
        if i % 1000 == 0:
            print('progress {:.2f}%'.format(float(100*i) / (len(prices) - hist - 1)))
        # The state is a `hist+2` dimensional vector. We'll force it to by a numpy matrix.
        current_state = np.asmatrix(np.hstack((prices[i:i+hist], budget, num_stocks)))
        # Calculate the portfolio value
        current_portfolio = budget + num_stocks * share_value
        # Select an action from the current policy
        action = policy.select_action(current_state, i)
        share_value = float(prices[i + hist])
        # Update portfolio values based on action
        if action == 'Buy' and budget >= share_value:
            budget -= share_value
            num_stocks += 1
            history.append((prices[i], 'BUY'))
            portfolio_history.append((current_portfolio, 'BUY'))
        elif action == 'Sell' and num_stocks > 0:
            budget += share_value
            num_stocks -= 1
            history.append((prices[i],'SELL'))
            portfolio_history.append((current_portfolio,'SELL'))
        else:
            action = 'Hold'
            history.append((prices[i], 'HOLD'))
            portfolio_history.append((current_portfolio, 'HOLD'))

        # Compute new portfolio value after taking action
        new_portfolio = budget + num_stocks * share_value
        if learn:
            reward = new_portfolio - current_portfolio
            next_state = np.asmatrix(np.hstack((prices[i+1:i+hist+1], budget, num_stocks)))
            transitions.append((current_state, action, reward, next_state))
            policy.update_q(current_state, action, reward, next_state)

    # Compute final portfolio worth
    portfolio = budget + num_stocks * share_value
    return portfolio, history, portfolio_history
```

In [74]:

```
def run_simulations(policy, budget, num_stocks, prices, hist):
    # Decide number of times to re-run the simulations
    num_tries = 5
    # Store portfolio worth of each run in this array
    final_portfolios = list()
    # Run this simulation
    #for _ in range(num_tries):
    #    portfolio, history, portfolio_history = run_simulation(policy, budget, num_stocks,
prices, hist)
    #    final_portfolios.append(portfolio)
    #    print('Final portfolio: ${}'.format(portfolio))
    #plt.title('Final Portfolio Value')
    #plt.xlabel('Epochs')
    #plt.ylabel('Training Portfolio Value')
    #plt.plot(final_portfolios)
    #plt.show()
```

In [75]:

```
if __name__ == "__main__":
    tf.reset_default_graph()
    prices = data.Close
    n = len(prices)
    n_train = int(n * 0.25)
    train_prices = prices[:n_train]
    test_prices = prices[n_train:]
```

```
    #print(train_prices)
    #print(test_prices)
    #plot_prices(train_prices, test_prices)

    # Define the list of actions the agent can take
    actions = ['Buy', 'Sell', 'Hold']
    hist = 200
    # Initial a decision policy
    policy = QLearningDecisionPolicy(actions, hist + 2)
    # Set the initial amount of money available to use
    budget = 1000.0
    # Set the number of stocks already owned
    num_stocks = 0
    # Run simulations multiple times to compute expected value of final net worth
    run_simulations(policy, budget, num_stocks, train_prices, hist)

    # Exacute on test set
    portfolio, history, portfolio_history = run_simulation(policy, budget, num_stocks, test_prices,
hist, learn=False)
    print(portfolio)
    print( np.std([x[0] for x in portfolio_history]))
```

```
progress 0.00%
progress 76.45%
1661.1001569999999
224.5073617232231
```

In [76]:

```
if __name__ == "__main__":
    prices = data.Close
    n = len(prices)
    n_train = int(n * 0.25)
    train_prices = prices[:n_train]
    test_prices = prices[n_train:]
    #print(train_prices)
    #print(test_prices)
    #plot_prices(train_prices, test_prices)

    # Define the list of actions the agent can take
    actions = ['Buy', 'Hold', 'Sell']
    # Initial a decision policy
    policy = RandomDecisionPolicy(actions)
    # Set the initial amount of money available to use
    budget = 1000.0
    # Set the number of stocks already owned
    num_stocks = 0
    # Run simulations multiple times to compute expected value of final net worth
    run_simulation(policy, budget, num_stocks, train_prices, hist)

    # Exacute on test set
    portfolio_rand, history_rand, portfolio_history_rand = run_simulation(policy, budget, num_stock
s, test_prices, hist, learn=False)
    print(portfolio_rand)
    print(np.std([x[0] for x in portfolio_history_rand]))
```

```
progress 0.00%
progress 0.00%
progress 76.45%
1442.4998770000016
108.26889616832723
```

In [77]:

```
if __name__ == "__main__":
    prices = data.Close
    n = len(prices)
    n_train = int(n * 0.25)
    train_prices = prices[:n_train]
    test_prices = prices[n_train:]
    #print(train_prices)
    #print(test_prices)
    #plot_prices(train_prices, test_prices)
```

```
    # Define the list of actions the agent can take
    actions = ['Buy', 'Hold', 'Hold']
    # Initial a decision policy
    policy = RandomDecisionPolicy(actions)
    # Set the initial amount of money available to use
    budget = 1000.0
    # Set the number of stocks already owned
    num_stocks = 0
    # Run simulations multiple times to compute expected value of final net worth
    run_simulation(policy, budget, num_stocks, train_prices, hist)

    # Exacute on test set
    portfolio_bh, history_bh, portfolio_history_bh = run_simulation(policy, budget, num_stocks, tes
t_prices, hist, learn=True)
    print(portfolio_bh)
    print(np.std([x[0] for x in portfolio_history_bh]))
```

```
progress 0.00%
progress 0.00%
progress 76.45%
1394.0
163.35898788143172
```

In [78]:

```
test_prices_plot = pd.DataFrame(test_prices)
test_prices_plot.index.name = 'Date'
test_prices_plot.reset_index(inplace=True)


# filter out the desired features
test_prices_plot = test_prices_plot[['Date', 'Close']]
# rename feature column names
test_prices_plot = test_prices_plot.rename(columns={'Close': 'actual', 'Date': 'date'})
# convert dates from object to DateTime type
dates = test_prices_plot['date']
dates = pd.to_datetime(dates, infer_datetime_format=True)
test_prices_plot['date'] = dates
test_prices_plot = test_prices_plot[hist:]
```

In [79]:

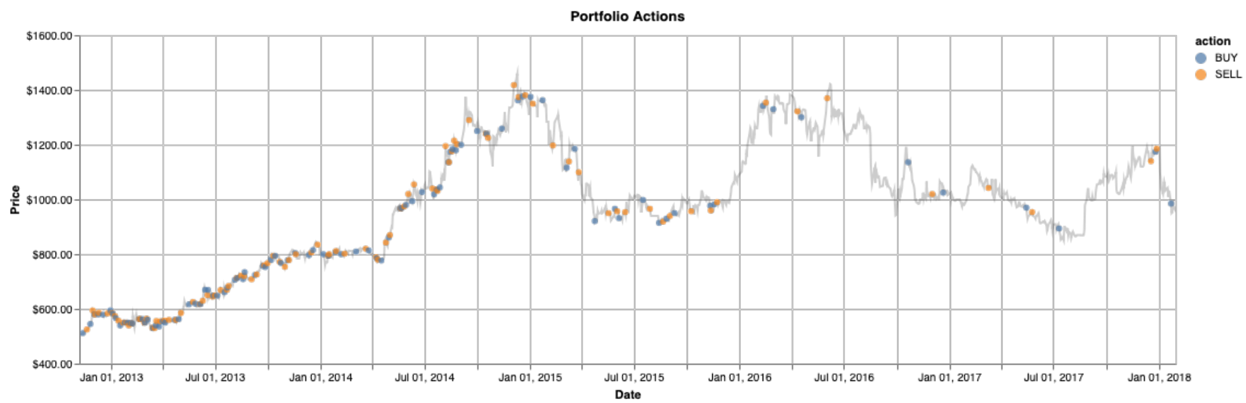
```
def visualize(df, history):
    # add history to dataframe
    position = [history[0][0]] + [x[0] for x in history]
    actions = ['HOLD'] + [x[1] for x in history]
    df['position'] = position
    df['action'] = actions

    # specify y-axis scale for stock prices
    scale = alt.Scale(domain=(min(min(df['actual']), min(df['position'])) - 50, max(max(df['actual'
]), max(df['position'])) + 50), clamp=True)

    # plot a line chart for stock positions
    actual = alt.Chart(df).mark_line(
        color='black',
        opacity=0.2
    ).encode(
        x='date:T',
        y=alt.Y('position', axis=alt.Axis(format='$.2f', title='Price'), scale=scale)
    ).interactive(
        bind_y=False
    )

    # plot the BUY and SELL actions as points
    points = alt.Chart(df).transform_filter(
        alt.datum.action != 'HOLD'
    ).mark_point(
        filled=True
    ).encode(
        x=alt.X('date:T', axis=alt.Axis(title='Date')),
        y=alt.Y('position', axis=alt.Axis(format='$.2f', title='Price'), scale=scale),
        color='action'
        #color=alt.Color('action', scale=alt.Scale(range=['blue', 'green', 'red']))
    ).interactive(bind_y=False)
```

```
    ).interactive(bind_y=False)

    # merge the two charts
    chart = alt.layer(actual, points, title="Portfolio Actions").properties(height=300, width=1000)

    return chart
```

In [80]:

```
chart = visualize(test_prices_plot, history)
chart
```

Out[80]:



In [81]:

```
def visualize2(df, history, portfolio_history, portfolio_history_rand, portfolio_history_bh):
    # add history to dataframe
    position = [history[0][0]] + [x[0] for x in history]
    portfolio_history =  [portfolio_history[0][0]] + [x[0] for x in portfolio_history]
    portfolio_history_rand =  [portfolio_history_rand[0][0]] + [x[0] for x in
portfolio_history_rand]
    portfolio_history_bh =  [portfolio_history_bh[0][0]] + [x[0] for x in portfolio_history_bh]
    actions = ['HOLD'] + [x[1] for x in history]
    df['position'] = position
    df['action'] = actions
    df['portfolio_history'] = portfolio_history
    df['portfolio_history_rand'] = portfolio_history_rand
    df['portfolio_history_bh'] = portfolio_history_bh


    # specify y-axis scale for stock prices
    scale = alt.Scale(domain=(min(min(df['portfolio_history_bh']), min(df['portfolio_history'])) -
50, max(max(df['portfolio_history_bh']), max(df['portfolio_history'])) + 50), clamp=True)


    # plot a line chart for stock positions
    actual = alt.Chart(df).mark_line(
        color='green',
        opacity=0.5
    ).encode(
        y=alt.Y('portfolio_history', axis=alt.Axis(format='$.2f', title='Portfolio Value'), scale=s
cale),
        x=alt.X('date:T', axis=alt.Axis(title='Date'))
    ).interactive(
        bind_y=False
    )

      # plot a line chart for stock positions
    actual_rand = alt.Chart(df).mark_line(
        color='blue',
        opacity=0.5
    ).encode(
        x='date:T',
        y=alt.Y('portfolio_history_rand', axis=alt.Axis(format='$.2f', title='Portfolio Value'))
    ).interactive(
        bind_y=False
    )
```

```
    # plot a line chart for stock positions
    actual_bh = alt.Chart(df).mark_line(
        color='red',
        opacity=0.5
    ).encode(
        x='date:T',
        y=alt.Y('portfolio_history_bh', axis=alt.Axis(format='$.2f', title='Portfolio Value'))
    ).interactive(
        bind_y=False
    )

    # merge the two charts
    chart = alt.layer(actual,actual_rand,actual_bh, title="Portfolio Value").properties(height=300,
width=1000)

    return chart
```

In [82]:

```
chart1 = visualize2(test_prices_plot, history, portfolio_history, portfolio_history_rand,
portfolio_history_bh)
chart1
```

Out[82]: