1. Libraries & Sample Data

The first step is to load our Python Libraries and download the sample data. The dataset represents Apple stock price (1d bars) for the year 2010

In [1]:
```
#!pip install --upgrade pip
#!pip install yfinance==0.2.54
#!pip install pandas_ta
```

In [2]:
```
# Load Python Libraries
import math
import keras
import random
import datetime
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
import pandas_ta as ta

from tqdm.notebook import tqdm
from collections import deque
from IPython.display import display, HTML
from sklearn.preprocessing import StandardScaler

# for dataframe display
pd.set_option('display.max_rows', None)
def display_df(df):
    # Puts the scrollbar next to the DataFrame
    display(HTML("<div style='height: 200px; overflow: auto; width: fit-cont

# for reproducability of training rounds
keras.utils.set_random_seed(42)
```

In [3]:
```
# Download Sample Data
#data = pd.read_csv('GOOG_2009-2010_6m_RAW_1d.csv')
data = yf.download('AAPL', start='2011-01-01', end='2011-06-30', interval='1
```

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

2. Exploratory Data Analysis

Next, we want to analyze our data. Display the data as a dataframe, and plot some relevant data so you can get an idea of what our dataset looks like.

In [4]:
```
# Display as Dataframe
display_df(data)
data.shape
```

	Close	High	Low	Open	Volume
Date					
2011-01-03	9.917945	9.938710	9.775603	9.799677	445138400
2011-01-04	9.969707	10.006119	9.875212	10.004314	309080800
2011-01-05	10.051261	10.061493	9.915840	9.917345	255519600
2011-01-06	10.043137	10.088878	10.018159	10.072929	300428800
2011-01-07	10.115060	10.121981	9.988064	10.050960	311931200

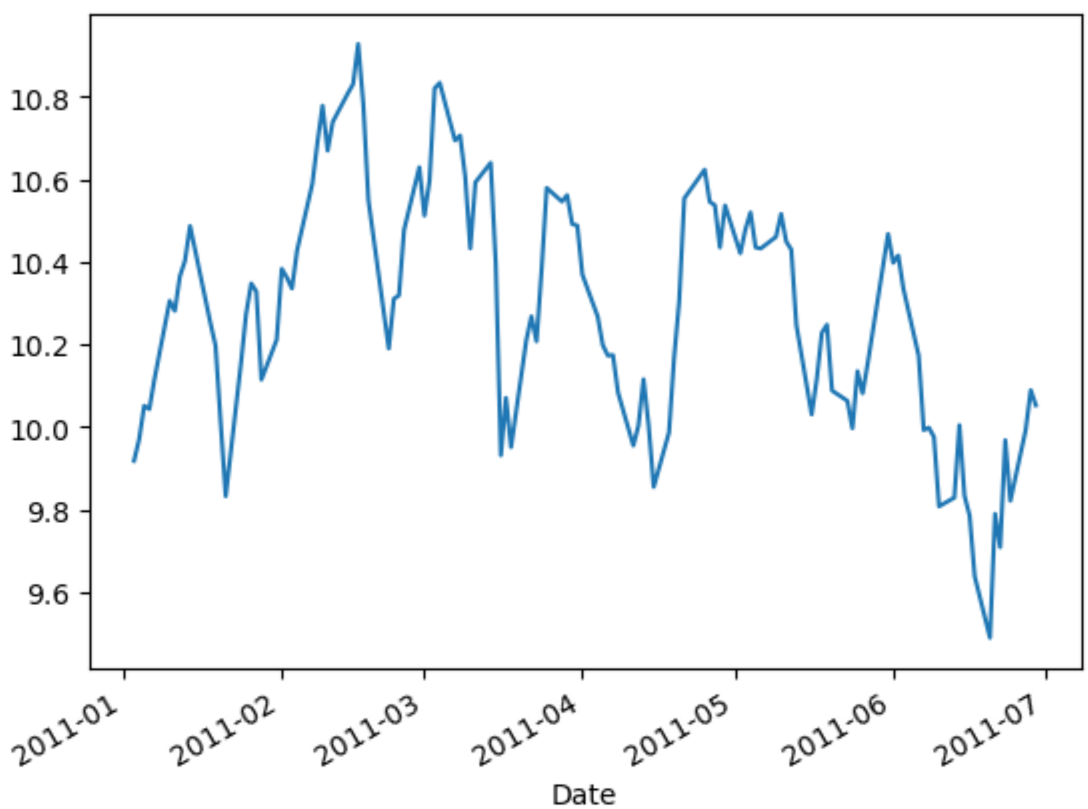
Out[4]:
```
(124, 5)
```

In [5]:
```
# Index data by Date
print(data.index)
```

```
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14',
               ...
               '2011-06-16', '2011-06-17', '2011-06-20', '2011-06-21',
               '2011-06-22', '2011-06-23', '2011-06-24', '2011-06-27',
               '2011-06-28', '2011-06-29'],
              dtype='datetime64[ns]', name='Date', length=124, freq=None)
```

In [6]:
```
# Plot the Close Data
data['Close'].plot()
```

Out[6]:
```
<Axes: xlabel='Date'>
```



3. Data Cleaning

Next, we need to clean our data for training our model. This requires removal of NaN values.

In [7]:
```
# Check for null values
print('Number of null values: ', data.isnull().sum())
```

```
Number of null values:  Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```

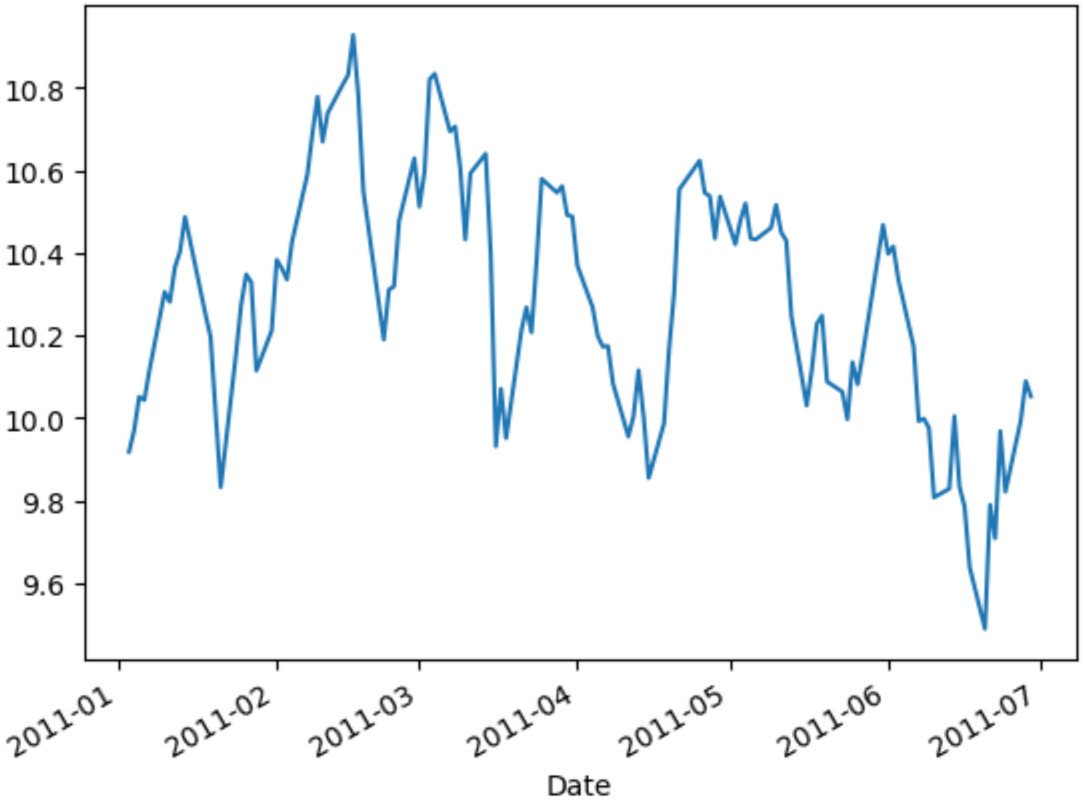
In [8]:
```
# forward fill missing values
data=data.ffill()
```

In [9]:
```
# Check for null values
print('Number of Null Values =', data.isnull().sum())
```

```
Number of Null Values = Close     0
High      0
Low       0
Open      0
Volume    0
dtype: int64
```

In [10]:
```
# Plot the cleaned Close Data
data['Close'].plot()
```

Out[10]:
```
<Axes: xlabel='Date'>
```



4. Feature Selection

Now that we have cleaned our stock data, we need to select which features to train our model on. For this project, we will be training with Close data and 20-day Bollinger Bands of Close.

In [11]:
```
# Calculate 20-day bollinger bands
data['MA5']= data['Close'].rolling(window=5).mean()
data['MA20'] = data['Close'].rolling(window=20).mean()
data['STD20'] = data['Close'].rolling(window=20).std()
data['BB_upper'] = data['MA20'] + (data['STD20'] * 2)
data['BB_lower'] = data['MA20'] - (data['STD20'] * 2)
data['Log_Ret'] = np.log(data['Close'] / data['Close'].shift(1))
data['Vol20'] = data['Log_Ret'].rolling(window=20).std() * np.sqrt(252)
data.ta.adx(append=True)
data.ta.atr(append=True)
data.ta.macd(append=True)
data.ta.obv(append=True)
data.ta.rsi(append=True)
display_df(data)
```

	Close	High	Low	Open	Volume	MA5
Date						
2011-01-03	9.917945	9.938710	9.775603	9.799677	445138400	NaN
2011-01-04	9.969707	10.006119	9.875212	10.004314	309080800	NaN
2011-01-05	10.051261	10.061493	9.915840	9.917345	255519600	NaN
2011-01-06	10.043137	10.088878	10.018159	10.072929	300428800	NaN
2011-01-07	10.115060	10.121981	9.988064	10.050960	311931200	10.019422

In [12]:
```
# Remove rows with NaN bollinger bands
data=data.dropna(axis=0)
```

In [13]:
```
# Define new dataframe with only the training features (Close, Upper BB, Low
dataset = data.reset_index()[['Date', 'Close', 'Volume', 'MA20', 'BB_upper',
```

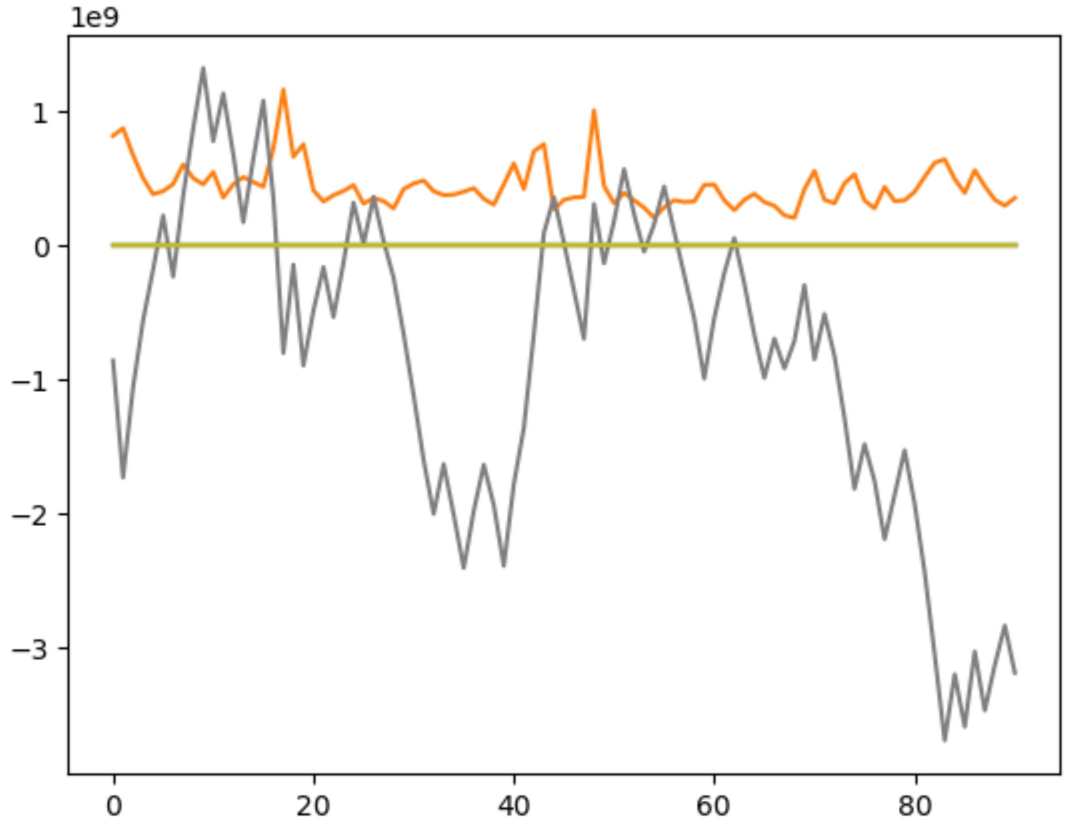
5. Normalization

Now that we have cleaned our data, created our indicators of interest, and selected our features, we must normalize our data. For this project, we use the sklearn StandardScaler, which centers the data and normalizes to unit variance. We will not be using a rolling scaler for this project, due to the complexity of back-translating to true proce and indicator values - you can try this yourself once you have completed the project.

In [14]:
```
# Display & Plot Un-normalized Dataset
display_df(dataset)
dataset['Close'].plot()
dataset['Volume'].plot()
dataset['MA20'].plot()
dataset['BB_upper'].plot()
dataset['BB_lower'].plot(rot=45)
dataset['Vol20'].plot()
dataset['MACD_12_26_9'].plot()
dataset['OBV'].plot()
dataset['RSI_14'].plot()
```

	Date	Close	Volume	MA20	BB_upper	BB_lower	Vol20
0	2011-02-18	10.549613	816057200	10.515185	11.016216	10.014154	0.206194
1	2011-02-22	10.189994	872555600	10.516930	11.012897	10.020964	0.218866
2	2011-02-23	10.310668	671854400	10.518766	11.011205	10.026327	0.218961
3	2011-02-24	10.318491	499900800	10.517306	11.012048	10.022564	0.217431
4	2011-02-25	10.477387	380018800	10.524754	11.011951	10.037557	0.224047
5	2011-02-28	10.629358	403074000	10.550499	10.999312	10.101686	0.213626

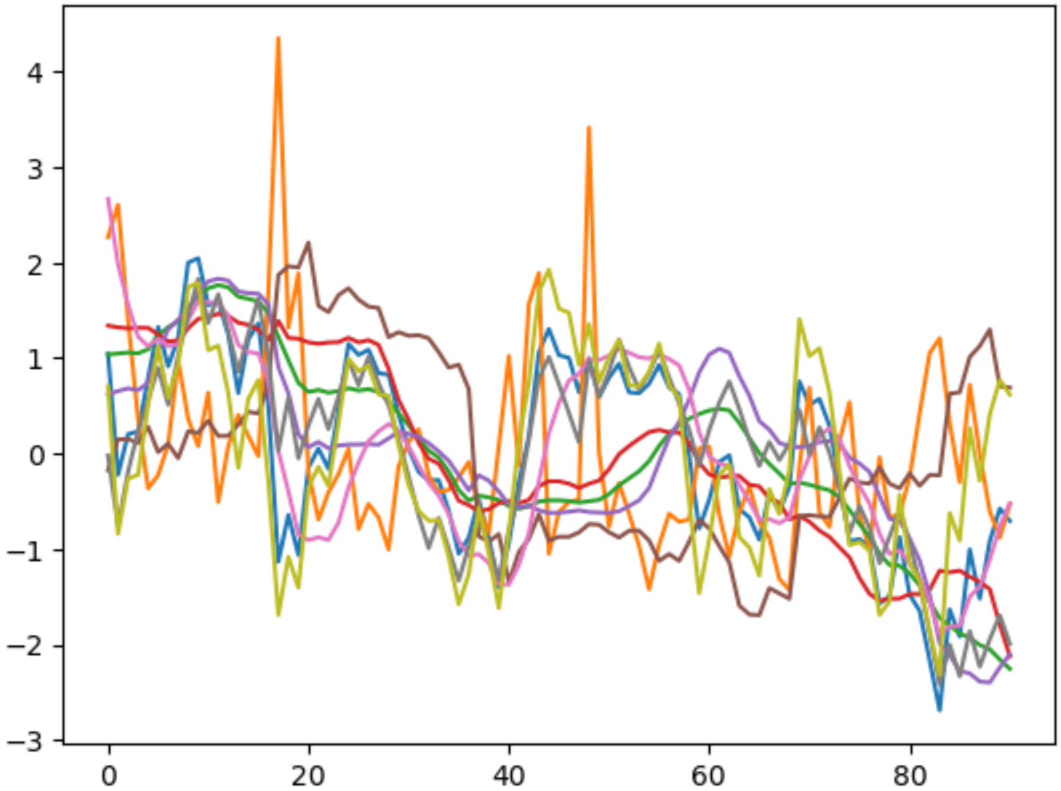
Out[14]: <Axes: >



In [15]:
```
# Normalize Dataset with StandardScaler
normlist = []
static_normed_dataset = pd.DataFrame(index=dataset.index)
for col in dataset.columns:
    if col == 'Date':
        static_normed_dataset[col] = dataset[col]
        continue
    normalizer = StandardScaler()
    column_data = pd.DataFrame(dataset[col])
    # fit normalizer to column data
    normalizer.fit(column_data)
    # transform column data with the fitted normalizer, and place the transf
    static_normed_dataset[col] = normalizer.transform(column_data).flatten()
    # append the fitted normalizer to normlist for use later
    normlist.append(normalizer)
```

In [16]:
```
# Display & Plot Normalized Dataset
static_normed_dataset['Close'].plot()
static_normed_dataset['Volume'].plot()
static_normed_dataset['MA20'].plot()
static_normed_dataset['BB_upper'].plot()
static_normed_dataset['BB_lower'].plot()
static_normed_dataset['Vol20'].plot()
static_normed_dataset['MACD_12_26_9'].plot()
static_normed_dataset['OBV'].plot()
static_normed_dataset['RSI_14'].plot()
```

Out[16]: <Axes: >



6. Train / Test Split

Now that our data cleaned, features are selected, and the dataset is normalized, we are ready to feed the data into our model. With this in mind, we split the data ito train and test data (50/50 split)

In [17]:
```
# split dataset df into train (50%) and test (50%) datasets
training_rows = int(len(static_normed_dataset.index)*0.5)
train_df = static_normed_dataset.loc[:training_rows].set_index("Date") # def
test_df = static_normed_dataset.loc[training_rows+1:].set_index("Date") # de
```

In [18]:
```
# display train and test dfs (ensure no overlap)
display_df(train_df)
display_df(test_df)
```

Date	Close	Volume	MA20	BB_upper	BB_lower	Vol20	ADX
2011-02-18	1.046374	2.266256	1.038592	1.340287	0.620369	-0.173404	-2.157
2011-02-22	-0.219733	2.606902	1.048491	1.322269	0.655763	0.150141	-1.733
2011-02-23	0.205123	1.396814	1.058901	1.313086	0.683634	0.152551	-1.355
2011-02-24	0.232665	0.360053	1.050622	1.317664	0.664077	0.113505	-1.046
2011-02-25	0.792090	-0.362752	1.092865	1.317138	0.742005	0.282415	-0.953

Date	Close	Volume	MA20	BB_upper	BB_lower	Vol20	AD
2011-04-27	1.002927	-0.506283	-0.495404	-0.313940	-0.607417	-0.870619	0.87
2011-04-28	0.642688	-0.477668	-0.511449	-0.359955	-0.592772	-0.823862	0.51
2011-04-29	1.000818	3.413560	-0.497623	-0.314401	-0.611044	-0.736271	0.33
2011-05-02	0.592898	0.015254	-0.482945	-0.281510	-0.615630	-0.743482	0.12
2011-05-03	0.796334	-0.764730	-0.423120	-0.163020	-0.619423	-0.822857	-0.07

In [19]:
```
# convert train and test dfs to np arrays with dtype=float
X_train = train_df.values.astype(float)# define training array under this va
X_test = test_df.values.astype(float)# define testing array under this varia
# print the shape of X_train to remind yourself how many examples and featur
print(X_train.shape)
print(X_test.shape)
# track index to remember which feature is which
idx_close = 0 # numerical idx of close data column in array
idx_volume = 1
idx_ma20 = 2
idx_bb_upper = 3 # numerical idx of BB Upper data column in array
idx_bb_lower = 4 # numerical idx of BB Upper data column in array
idx_vol20 = 5
idx_adx_14 = 6
idx_atrr_14 = 7
idx_macd_12_26_9 = 8
idx_obv = 9
idx_rsi_14 = 10
```

```
(46, 11)
(45, 11)
```

7. Define the Agent

Now that our data is ready to use, we can define the Reinforcement Learning Agent.

Define the DQN Model

The first step in defining our agent is the Deep Q-Network model definition. For this project, we are creating a model sequential model with four layers. The first three layers have output shape of 64, 32, and 8, respectively, and a RELU activation. The output layer has an output shape of the size of our action space (buy, sell, hold), and a linear activation. Our Loss finction is Mean Squared Error, and our optimizer is Adam with a learning rate of 0.001. Use Keras to build this model.

In [20]:
```
@keras.saving.register_keras_serializable()
# Define DQN Model Architecture
class DQN(keras.Model):
    def __init__(self, state_size, action_size):
        # define model layers in keras
        model = keras.models.Sequential()
        model.add(keras.layers.Dense(units=64, input_dim=state_size, activat
        #model.add(keras.layers.PReLU())
        model.add(keras.layers.Dense(units=32, activation="relu"))
        #model.add(keras.layers.PReLU())
        model.add(keras.layers.Dense(units=8, activation="relu"))
        #model.add(keras.layers.PReLU())
        model.add(keras.layers.Dense(action_size, activation="linear"))
        # compile model in keras
        model.compile(loss="mse", optimizer=keras.optimizers.Adam(learning_r
        # save model to DQN instance
        self.model = model
```

Define Agent Class

Now that we have defined our underlying DQN Model, we must define out Reinforcement Learning Agent. The agent initialization is provided for you, you must define an act function, and an expereince replay function. As a reminder, the act function defines how our model will act (buy, hold, or sell) given a certain state. The Experience Replay function tackles catastrophic forgetting in our training process, by maintaining a memory buffer to allow training on independent / randomized minibatches of previous states.

In [21]:
```
class Agent:
    def __init__(self, window_size, num_features, test_mode=False, model_nam
        self.window_size = window_size # How many days of historical data do
        self.num_features = num_features # How many training features do we
        self.state_size = window_size*num_features # State size includes num
        self.action_size = 3 # 0=hold, 1=buy, 2=sell
        self.memory = deque(maxlen=1000) # Bound memory size: once the memor
        self.inventory = [] # Inventory to hold trades
        self.model_name = model_name # filename for saved model checkpoint l
        self.test_mode = test_mode # flag for testing (allows model load fro

        self.gamma = 0.95
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995

        self.model = keras.models.load_model(model_name) if test_mode else s
```

```
    #Deep Q Learning (DQL) model
    def _model(self):
        model = DQN(self.state_size, self.action_size).model
        return model

    # DQL Predict (with input reshaping)
    #   Input = State
    #   Output = Q-Table of action Q-Values
    def get_q_values_for_state(self, state):
        return self.model.predict(state.flatten().reshape(1, self.state_size


    # DQL Fit (with input reshaping)
    #   Input = State, Target Q-Table
    #   Output = MSE Loss between Target Q-Table and Actual Q-Table for Stat
    def fit_model(self, input_state, target_output):
        return self.model.fit(input_state.flatten().reshape(1, self.state_si

    # Agent Action Selector
    #   Input = State
    #   Policy = epsilon-greedy (to minimize possibility of overfitting)
    #   Intitially high epsilon = more random, epsilon decay = less random l
    #   Output = Action (0, 1, or 2)
    def act(self, state):
        # Choose any action at random (Probablility = epsilon for training m
        if not self.test_mode and random.random() <= self.epsilon:
            return random.randrange(self.action_size)
        # Choose the action which has the highest Q-value (Probablitly = 1-e
        # **use model to select action here - i.e. use model to assign q-val
        # **return the action that has the highest value from the q-value fu
        options = self.get_q_values_for_state(state)
        return np.argmax(options[0])


    # Experience Replay (Learning Function)
    #   Input = Batch of (state, action, next_state) tuples
    #   Optimal Q Selection Policy = Bellman equation
    #   Important Notes = Model fitting step is in this function (fit_model)
    #                     Epsilon decay step is in this function
    #   Output = Model loss from fitting step
    def exp_replay(self, batch_size):
        losses = []
        # define a mini-batch which holds batch_size most recent previous me
        mini_batch = []
        l = len(self.memory)
        for i in range(l-batch_size + 1, l):
            mini_batch.append(self.memory[i])

        for state, action, reward, next_state, done in mini_batch:
            # reminders:
            #   - state is a vector containing close & MA values for the cur
            #   - action is an integer representing the action taken by the
            #   - reward represents the profit of a given action - it is eit
            #   - next_state is a vector containing close & MA values for th
            #   - done is a boolean flag representing whether or not we are
```

```
            if done:
                # special condition for last training epoch in batch (no nex
                optimal_q_for_action = reward
            else:
                # target Q-value is updated using the Bellman equation: rewa
                optimal_q_for_action = reward + self.gamma * np.max(self.get
            # Get the predicted Q-values of the current state
            target_q_table = self.get_q_values_for_state(state)
            # Update the output Q table - replace the predicted Q value for
            target_q_table[0][action] = optimal_q_for_action
            # Fit the model where state is X and target_q_table is Y
            history = self.fit_model(state, target_q_table)
            losses += history.history['loss']

        # define epsilon decay (for the act function)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
        return losses
```

8. Train the Agent

Now that our data is ready and our agent is defined, we are ready to train the agent.

Helper Functions

Before we define the training loop, we will write some helper functions: one for printing price data, one to define the sigmoid funtion, one to grab the state representation, one to plot the trading output of our trained model, and one to plot the training loss. The printing, sigmoid, and plotting functions are defined for you. You must define the function which gets the state representation.

In [22]:

```
# Format price string
def format_price(n):
    return ('-$' if n < 0 else '$') + '{0:.2f}'.format(abs(n))

def sigmoid(x):
    return 1 / (1 + math.exp(-x))

# Plot behavior of trade output
def plot_behavior(data_input, bb_upper_data, bb_lower_data, states_buy, stat
    fig = plt.figure(figsize = (15,5))
    plt.plot(data_input, color='k', lw=2., label= 'Close Price')
    plt.plot(bb_upper_data, color='b', lw=2., label = 'Bollinger Bands')
    plt.plot(bb_lower_data, color='b', lw=2.)
    plt.plot(data_input, '^', markersize=10, color='r', label = 'Buying sign
    plt.plot(data_input, 'v', markersize=10, color='g', label = 'Selling sig
    plt.title('Total gains: %f'%(profit))
    plt.legend()
    if train:
        plt.xticks(range(0, len(train_df.index.values), int(len(train_df.ind
    else:
```

```
        plt.xticks(range(0, len(test_df.index.values), int(len(test_df.index
    plt.show()

# Plot training loss
def plot_losses(losses, title):
    plt.plot(losses)
    plt.title(title)
    plt.ylabel('MSE Loss Value')
    plt.xlabel('batch')
    plt.show()

# returns an an n-day state representation ending at time t
def get_state(data, t, n):
    # data is the dataset of interest which holds the state values (i.e. Clo
    # t is the current time step
    # n is the size of the training window
    d = t - n
    if d >= 0:
        block = data[d:t]
    else:
        block =  np.array([data[0]]*n)
    # the first step is to get the window of the dataset at the current time
    # remember to define the special case for the first iteration, where the
    res = []
    for i in range(n - 1):
        feature_res = []
        for feature in range(data.shape[1]):
            feature_res.append(sigmoid(block[i + 1, feature] - block[i, feat
        res.append(feature_res)
    # once we have our state data, we need to apply the sigmoid to each feat
    # return an array holding the n-day sigmoid state representation
    return np.array([res])
```

Training Loop

In [23]:
```
# display the shape of your training data in order to remond yourself how ma
X_train.shape
```

Out[23]: (46, 11)

In [24]:
```
keras.utils.disable_interactive_logging()  # disable built-in keras loading

window_size = 1

agent = Agent(window_size, num_features=X_train.shape[1])# instatntiate the a
```

```
/Users/ishanklal/miniconda3/envs/aitsnd/lib/python3.12/site-packages/keras/s
rc/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input
_dim` argument to a layer. When using Sequential models, prefer using an `In
put(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

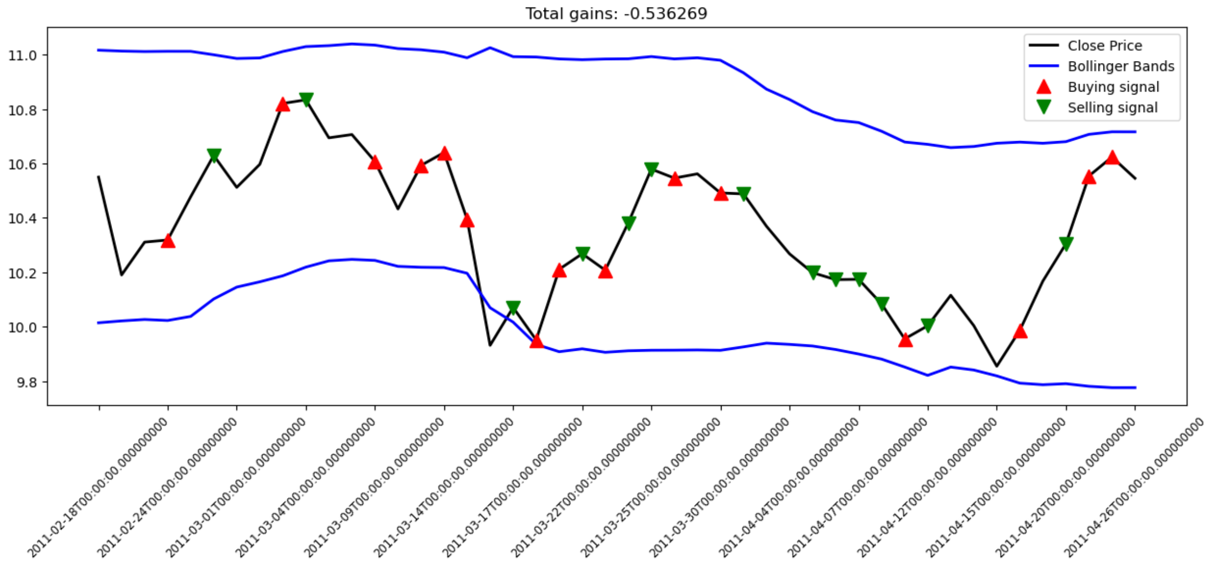
In [25]:
```
keras.config.disable_traceback_filtering() # disable built-in keras loading

l = X_train[:, 0].shape[0]-1 # track number of examples in dataset (i.e. num
```

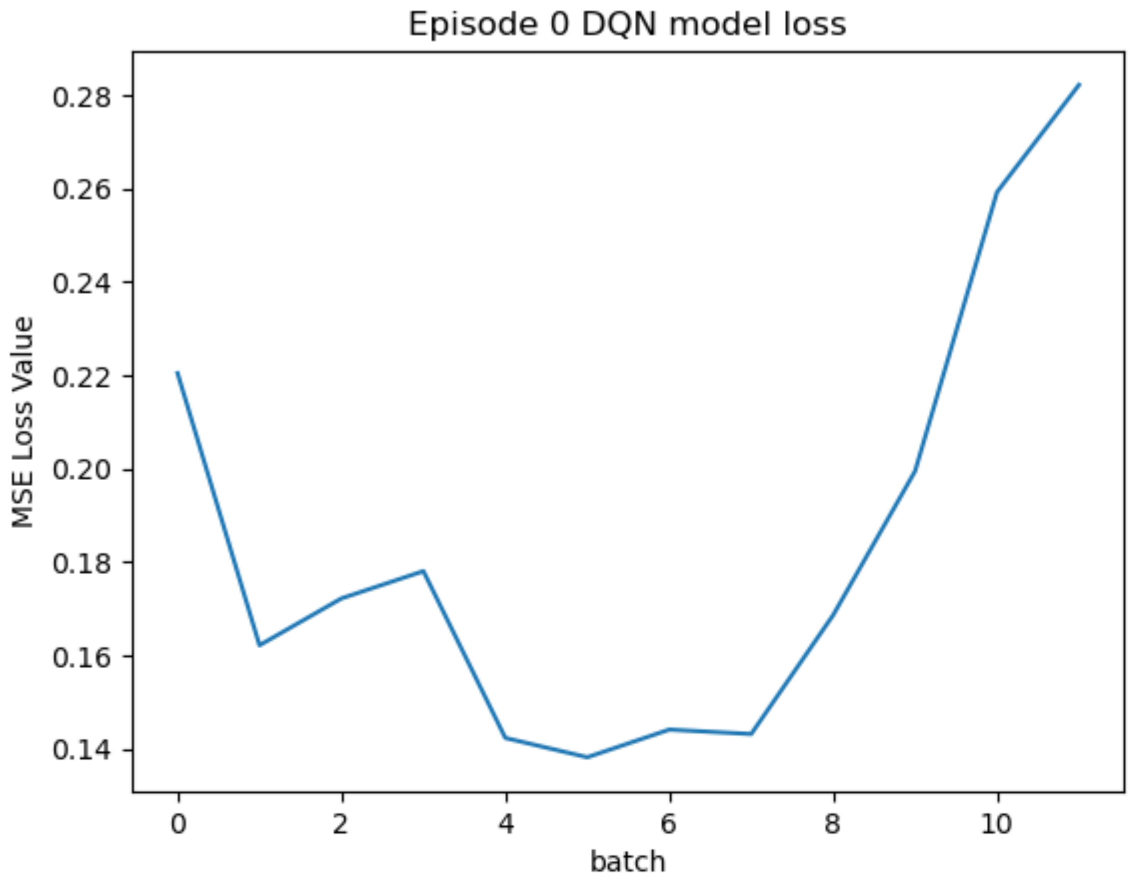
```
# batch size defines how often to run the exp_replay method
batch_size = 32

#An episode represents a complete pass over the data.
episode_count = 2

normalizer_close = normlist[idx_close]       # get the close normalizer from
normalizer_bb_upper = normlist[idx_bb_upper] # get the BB upper normalizer f
normalizer_bb_lower = normlist[idx_bb_lower] # get the BB lower normalizer f

X_train_true_price = normalizer_close.inverse_transform(X_train[:, idx_close
X_train_true_bb_upper = normalizer_bb_upper.inverse_transform(X_train[:, idx
X_train_true_bb_lower = normalizer_bb_lower.inverse_transform(X_train[:, idx

batch_losses = []
num_batches_trained = 0

for e in range(episode_count + 1):
    state = get_state(X_train, 0, window_size + 1) # get the state for the f
    # initialize variables
    total_profit = 0
    total_winners = 0
    total_losers = 0
    agent.inventory = []
    states_sell = []
    states_buy = []
    for t in tqdm(range(l), desc=f'Running episode {e}/{episode_count}'):
        # get the action
        action = agent.act(state)
        # get the next state
        next_state = get_state(X_train, t+1, window_size + 1)
        # initialize reward for the current time step
        reward = 0

        if action == 1: # buy
            # inverse transform to get true buy price in dollars
            buy_price = X_train_true_price[t, idx_close]
            # append the buy price to the inventory
            agent.inventory.append(buy_price)
            # append the time step to states_buy
            states_buy.append(t)
            # print the action and price of the action
            print(f'Buy: {format_price(buy_price)}')

        elif action == 2 and len(agent.inventory) > 0: # sell
            # get the bought price of the stock you are selling (i.e. the st
            bought_price = agent.inventory.pop(0)
            # inverse transform to get true sell price in dollars
            sell_price = X_train_true_price[t, idx_close]
            # define reward as max of profit (close price at time of sell -
            trade_profit = sell_price - bought_price
            reward = max(trade_profit, 0)
            total_profit += trade_profit
            # add current profit to total profit
            if trade_profit >=0:
```

```
                # add current profit to total winners
                total_winners += trade_profit
            else:
                # add current profit to total losers
                total_losers += trade_profit
            # append the time step to states_sell
            states_sell.append(t)
            # print the action, price of the action, and profit of the actio
            print(f'Sell: {format_price(sell_price)} | Profit: {format_price

        # flag for final training iteration
        done = True if t == l - 1 else False

        # append the details of the state action etc in the memory, to be us
        agent.memory.append((state, action, reward, next_state, done))
        state = next_state
        # print total profit and plot behaviour of the current episode when
        if done:
            print('--------------------------------')
            print(f'Episode {e}')
            print(f'Total Profit: {format_price(total_profit)}')
            print(f'Total Winners: {format_price(total_winners)}')
            print(f'Total Losers: {format_price(total_losers)}')
            print(f'Max Loss: {max(batch_losses[num_batches_trained:len(batc
            print(f'Total Loss: {sum(batch_losses[num_batches_trained:len(ba
            print('--------------------------------')
            plot_behavior(X_train_true_price, X_train_true_bb_upper, X_train
            plot_losses(batch_losses[num_batches_trained:len(batch_losses)],
            num_batches_trained = len(batch_losses)

        # when the size of the memory is greater than the batch size, run th
        # then sum the losses for the batch and append them to the batch_los
        if len(agent.memory) > batch_size:
            losses = agent.exp_replay(batch_size)
            batch_losses.append(sum(losses))

    if e % 2 == 0:
        # save the model every 2 episodes (in case of crash or better traini
        agent.model.save(f'model_ep{e}.keras')
```

```
Running episode 0/2:   0%|          | 0/45 [00:00<?, ?it/s]
```

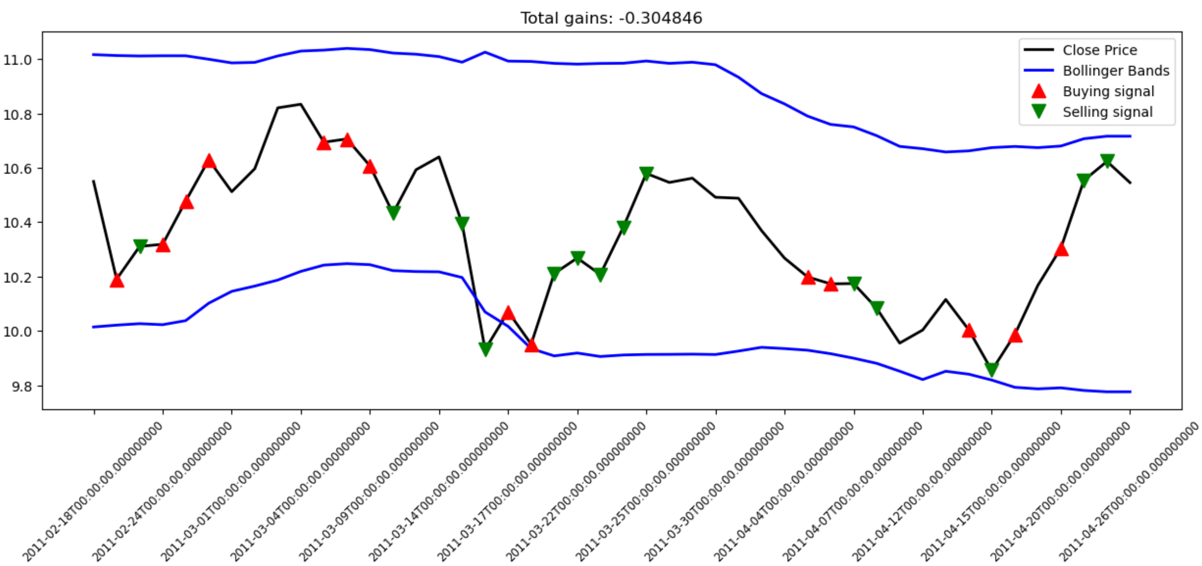
```
Buy: $10.32
Sell: $10.63 | Profit: $0.31
Buy: $10.82
Sell: $10.83 | Profit: $0.01
Buy: $10.61
Buy: $10.59
Buy: $10.64
Buy: $10.40
Sell: $10.07 | Profit: -$0.54
Buy: $9.95
Buy: $10.21
Sell: $10.27 | Profit: -$0.32
Buy: $10.21
Sell: $10.38 | Profit: -$0.26
Sell: $10.58 | Profit: $0.18
Buy: $10.55
Buy: $10.49
Sell: $10.49 | Profit: $0.54
Sell: $10.20 | Profit: -$0.01
Sell: $10.17 | Profit: -$0.03
Sell: $10.17 | Profit: -$0.37
Sell: $10.08 | Profit: -$0.41
Buy: $9.95
Sell: $10.00 | Profit: $0.05
Buy: $9.99
Sell: $10.30 | Profit: $0.32
Buy: $10.55
Buy: $10.62
--------------------------------
Episode 0
Total Profit: -$0.54
Total Winners: $1.41
Total Losers: -$1.95
Max Loss: 0.2821326352204778
Total Loss: 2.2092875352602164
--------------------------------
```

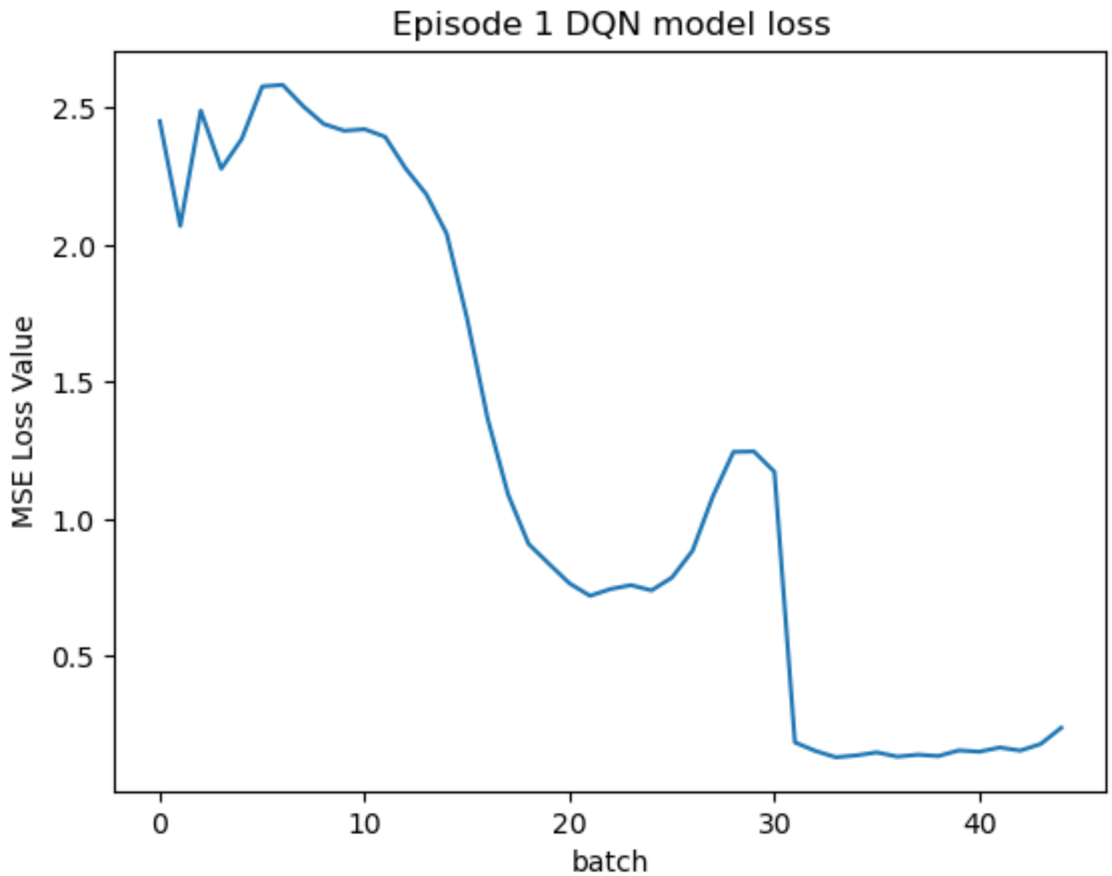




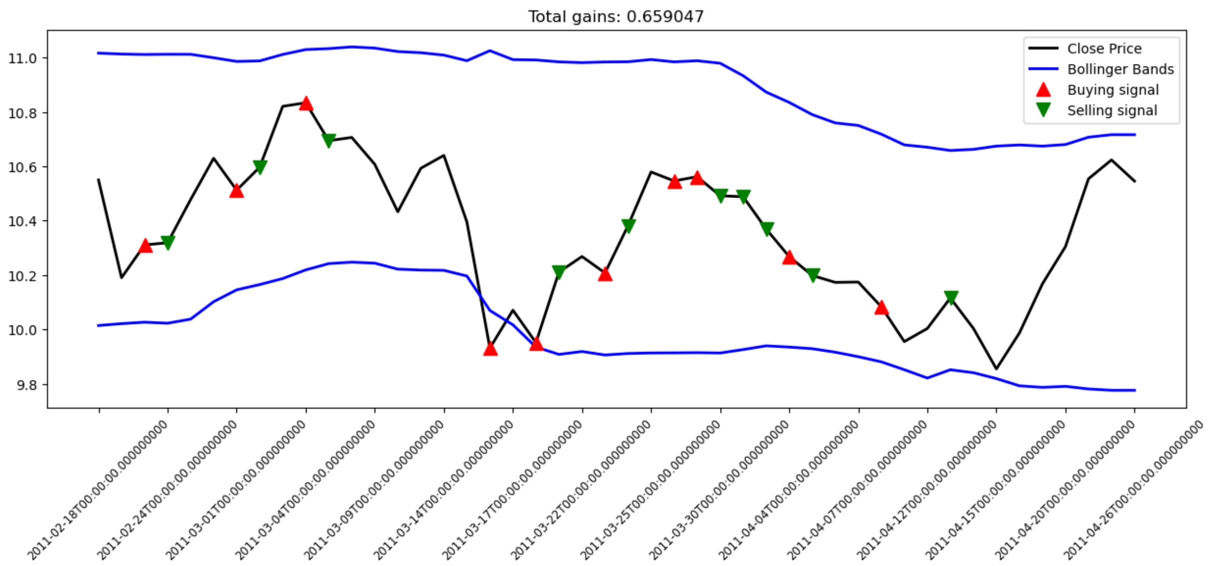

```
Running episode 1/2:   0%|          | 0/45 [00:00<?, ?it/s]
```

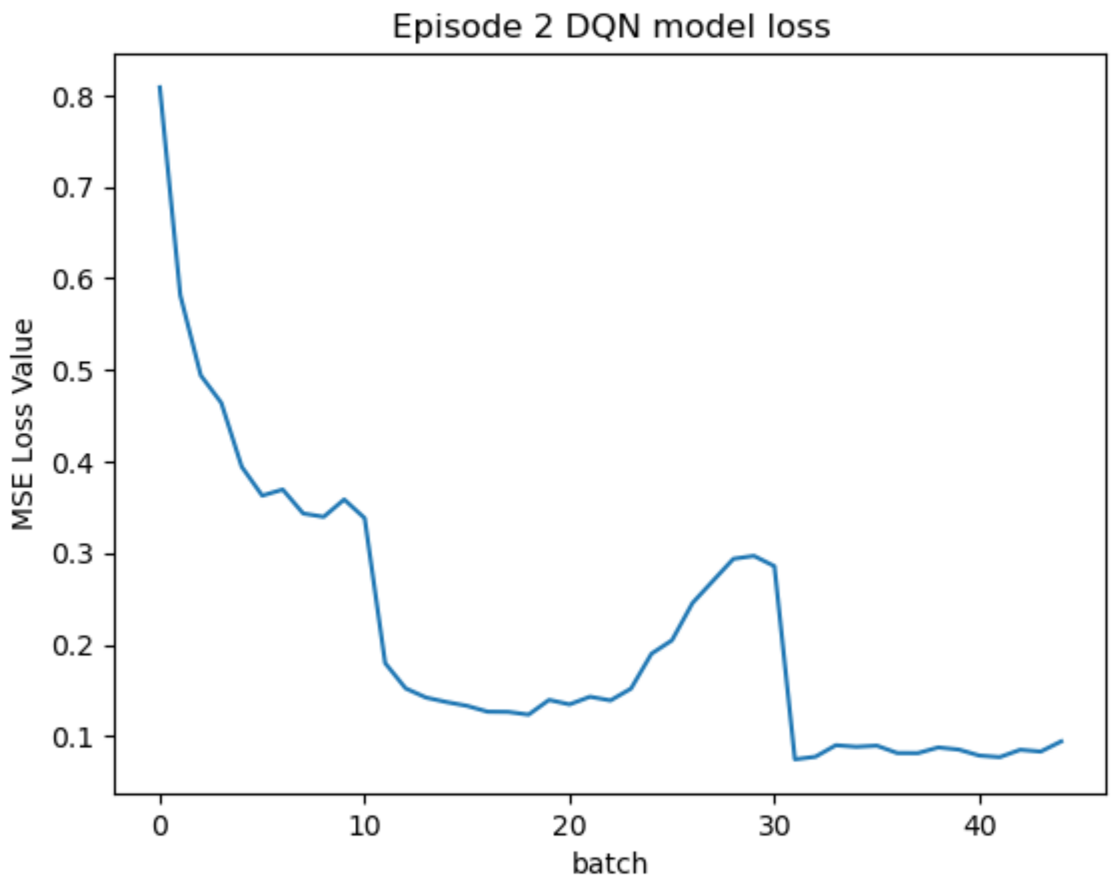
```
Buy: $10.19
Sell: $10.31 | Profit: $0.12
Buy: $10.32
Buy: $10.48
Buy: $10.63
Buy: $10.69
Buy: $10.71
Buy: $10.61
Sell: $10.43 | Profit: $0.11
Sell: $10.40 | Profit: -$0.08
Sell: $9.93 | Profit: -$0.70
Buy: $10.07
Buy: $9.95
Sell: $10.21 | Profit: -$0.48
Sell: $10.27 | Profit: -$0.44
Sell: $10.21 | Profit: -$0.40
Sell: $10.38 | Profit: $0.31
Sell: $10.58 | Profit: $0.63
Buy: $10.20
Buy: $10.17
Sell: $10.17 | Profit: -$0.02
Sell: $10.08 | Profit: -$0.09
Buy: $10.00
Sell: $9.85 | Profit: -$0.15
Buy: $9.99
Buy: $10.30
Sell: $10.55 | Profit: $0.57
Sell: $10.62 | Profit: $0.32
--------------------------------
Episode 1
Total Profit: -$0.30
Total Winners: $2.06
Total Losers: -$2.36
Max Loss: 2.5843674511289265
Total Loss: 53.81107894537623
--------------------------------
```





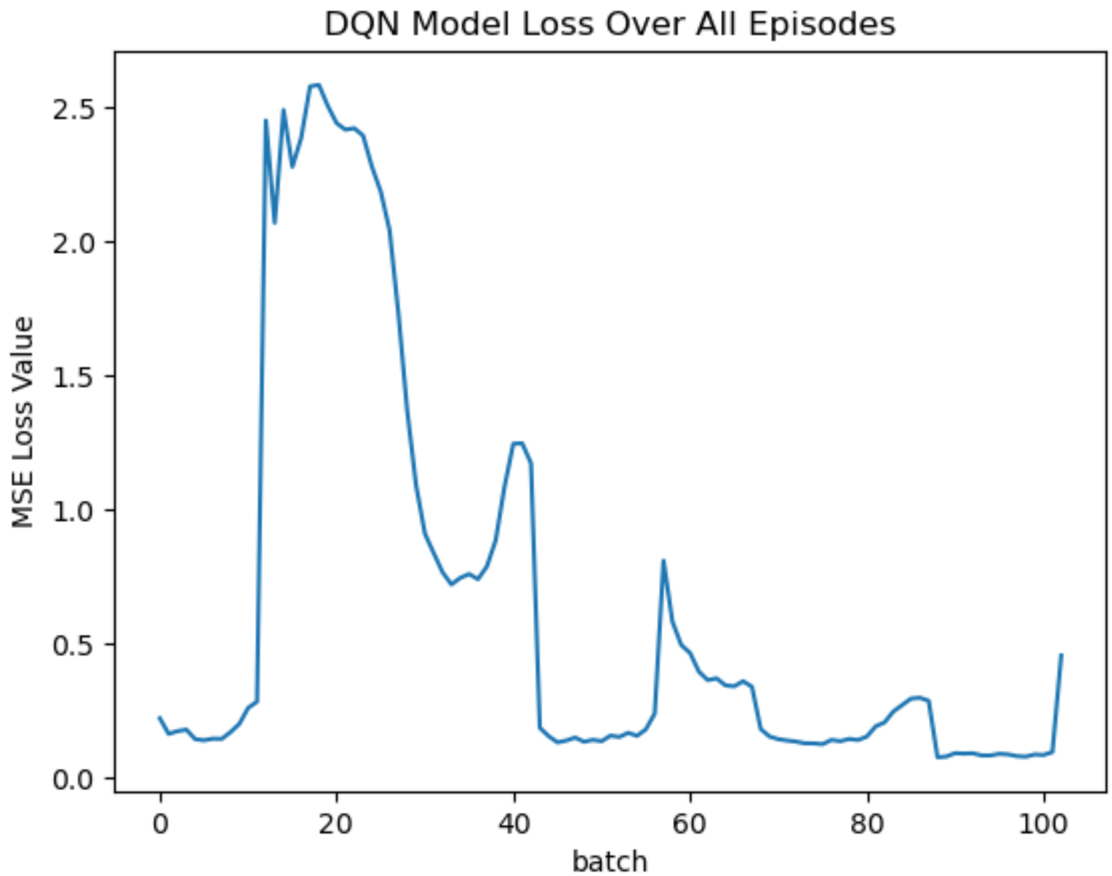

```
Running episode 2/2:   0%|          | 0/45 [00:00<?, ?it/s]
Buy: $10.31
Sell: $10.32 | Profit: $0.01
Buy: $10.51
Sell: $10.60 | Profit: $0.08
Buy: $10.83
Sell: $10.69 | Profit: -$0.14
Buy: $9.93
Buy: $9.95
Sell: $10.21 | Profit: $0.28
Buy: $10.21
Sell: $10.38 | Profit: $0.43
Buy: $10.55
Buy: $10.56
Sell: $10.49 | Profit: $0.28
Sell: $10.49 | Profit: -$0.06
Sell: $10.37 | Profit: -$0.19
Buy: $10.27
Sell: $10.20 | Profit: -$0.07
Buy: $10.08
Sell: $10.12 | Profit: $0.03
--------------------------------
Episode 2
Total Profit: $0.66
Total Winners: $1.12
Total Losers: -$0.46
Max Loss: 0.8085520108183317
Total Loss: 9.643077874005053
--------------------------------
```





Plot Training Loss

In [26]:
```
# use the plot_losses function to plot all batch_losses for the entire train
plot_losses(batch_losses, "DQN Model Loss Over All Episodes")
```



9. Test the trained agent

Finally, we get to test our trained model to see how well it performs in our test set. Using the training loop above, define a method to run our trained model on our X_test dataset.

Define Parameters

Some test parameters are defined for you below. Fill out the missing data. If you need a hint, look up at the training loop.

In [27]:

```
l_test = len(X_test) - 1
state = get_state(X_test, 0, window_size + 1)
total_profit = 0
done = False
states_sell_test = []
states_buy_test = []

#Get the trained model
agent = Agent(window_size, num_features=X_test.shape[1], test_mode=True, mod
agent.inventory = []

state = get_state(X_test, 0, window_size + 1) # get the first state of the t

X_test_true_price = normalizer_close.inverse_transform(X_test[:, idx_close].
```

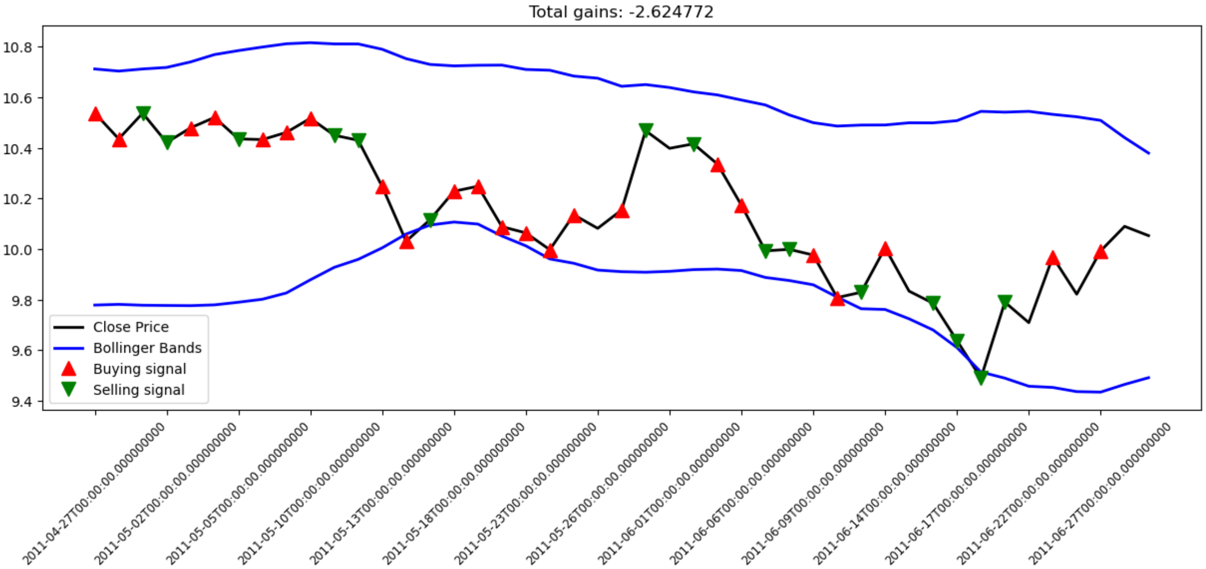
```
X_test_true_bb_upper = normalizer_bb_upper.inverse_transform(X_test[:, idx_b
X_test_true_bb_lower = normalizer_bb_lower.inverse_transform(X_test[:, idx_b
```

Run the Test

Run the test data through the trained model. Look at the training loop for a hint.

In [28]:
```
for t in range(l_test):
    action = agent.act(state)
    next_state = get_state(X_test, t + 1, window_size + 1) # get the next st
    reward = 0

    if action == 1: # buy
        # inverse transform to get true buy price in dollars
        buy_price = X_test_true_price[t, idx_close]
        # append buy prive to inventory
        agent.inventory.append(buy_price)
        # append time step to states_buy_test
        states_buy_test.append(t)
        print(f'Buy: {format_price(buy_price)}')

    elif action == 2 and len(agent.inventory) > 0: # sell
        # get bought price from beginning of inventory
        bought_price = agent.inventory.pop(0)
        # inverse transform to get true sell price in dollars
        sell_price = X_test_true_price[t, idx_close]
        # reward is max of profit (close price at time of sell - close price
        reward = max(sell_price - bought_price, 0)
        # update total_test_profit
        total_profit += sell_price - bought_price
        states_sell_test.append(t)
        # append time step to states_sell_test
        print(f'Sell: {format_price(sell_price)} | Profit: {format_price(sel


    if t == l_test - 1:
        done = True
    # append to memory so we can re-train on 'live' (test) data later
    agent.memory.append((state, action, reward, next_state, done))
    state = next_state

    if done:
        print('------------------------------------------')
        print(f'Total Profit: {format_price(total_profit)}')
        print('------------------------------------------')

plot_behavior(X_test_true_price, X_test_true_bb_upper, X_test_true_bb_lower,
```

```
Buy: $10.54
Buy: $10.43
Sell: $10.54 | Profit: -$0.00
Sell: $10.42 | Profit: -$0.01
Buy: $10.48
Buy: $10.52
Sell: $10.43 | Profit: -$0.04
Buy: $10.43
Buy: $10.46
Buy: $10.52
Sell: $10.45 | Profit: -$0.07
Sell: $10.43 | Profit: -$0.00
Buy: $10.25
Buy: $10.03
Sell: $10.12 | Profit: -$0.34
Buy: $10.23
Buy: $10.25
Buy: $10.09
Buy: $10.06
Buy: $10.00
Buy: $10.13
Buy: $10.15
Sell: $10.47 | Profit: -$0.05
Sell: $10.42 | Profit: $0.17
Buy: $10.34
Buy: $10.17
Sell: $9.99 | Profit: -$0.04
Sell: $10.00 | Profit: -$0.23
Buy: $9.98
Buy: $9.81
Sell: $9.83 | Profit: -$0.42
Buy: $10.00
Sell: $9.79 | Profit: -$0.30
Sell: $9.64 | Profit: -$0.43
Sell: $9.49 | Profit: -$0.51
Sell: $9.79 | Profit: -$0.35
Buy: $9.97
Buy: $9.99
------------------------------------------
Total Profit: -$2.62
------------------------------------------
```


Total gains: -2.624772
10.8
10.6
10.4
10.2
10.0
9.8
9.6
9.4
Close Price
Bollinger Bands
Buying signal
Selling signal
2011-04-27T00:00:00.000000000
2011-05-02T00:00:00.000000000
2011-05-05T00:00:00.000000000
2011-05-10T00:00:00.000000000
2011-05-13T00:00:00.000000000
2011-05-18T00:00:00.000000000
2011-05-23T00:00:00.000000000
2011-05-26T00:00:00.000000000
2011-06-01T00:00:00.000000000
2011-06-06T00:00:00.000000000
2011-06-09T00:00:00.000000000
2011-06-14T00:00:00.000000000
2011-06-17T00:00:00.000000000
2011-06-22T00:00:00.000000000
2011-06-27T00:00:00.000000000

In [ ]:

In [ ]:

In [ ]: