RL for Stocks

Harry Yin

October 2023

1 Motivations for RL in Forecasting Stock Prices

There were two main algorithms we could have done for our predictions of stock prices: Supervised Learning and Reinforcement Learning. Let us list out the benefits and downsides.

Supervised Learning: The pros are the fact that the training model is relatively easy. The learning speed of supervised learning usually surpasses that of a reinforcement learning algorithm and the model loss function is easier to play with. The con is that it can't be expanded upon. If I wanted it to learn further days with continuity, that would not be possible.

Reinforcement Learning: The major pro is the fact that it can be expanded upon. If I want it to predict the stock consistently for the next 2 weeks, it would be possible, unlike with supervised learning. However, unlike supervised learning, the training is much more difficult. Supervised learning produces results quicker and more consistently.

Verdict: All in all, I decided to go with reinforcement learning. The main reason was personal; I wanted to try out reinforcement learning. The training turned out to be a massive drawback, however, I decided the learning experience and the possibility of expansion outweighed that drawback.

2 Development of the Policy Algorithm

First of all, I'd like to thanks Andrej Karpathy's Blog for his assistance on the reinforcement learning algorithm: Please check his stuff out here: http://karpathy.github.io/2016/05/31/rl/ (Blog), https://gist.github.com/karpathy/a4166c7fe253700972fcbc77e4ea32c5 (Code).

Now, following Karpathy's code like a guidline, here is how this policy algorithm was formed. Like all models, we first begin with the model initialization.

```
def model_init(self):
    self.model = {
        1: np.random.randn(self.hidden_layers,
            self.num_states) / np.sqrt(self.num_actions)
            * self.learning_rate,
        2: np.random.randn(self.num_actions,
            self.hidden_layers) /
            np.sqrt(self.hidden_layers)
            * self.learning_rate,
        }
```

So, funny thing you might have noticed, after Xavier initialization, there is a multiplication by the learning rate. This, while I haven't seen them in any of the policy models done with only numpy, is a very important aspect of the code. It would be more accurate to say that the learning rate is important, as given some state, the learning rate too has to be scaled either up or down to match said state, thus why the multiplication by learning rate is necessary. Other than that, the shapes of the two models are once again classic, with the first layer of hidden layers by state size and the second layer of action size by hidden layer.

Thusly, we have policy forward.

```
def policy_forward(self, state):
    m1 = self.model[1]
    m2 = self.model[2]
    h = np.dot(self.model[1], state)
    h[h<0] = 0
    logp = np.dot(self.model[2], h)
    p = self.softmax(logp)
    return p, h
```

By doing a series of matrix products, we arrive flat array of values that we then flatten to get percentages of the actions based on value size relativity.

```
def policy_backward(self, processed_states,
    hiddle_layer_output, grad_log_prob):
    dW2 = np.dot(hiddle_layer_output.T, grad_log_prob).T
    dh = np.dot(grad_log_prob, self.model[2])
    dh[hiddle_layer_output <= 0] = 0 # backprop relu
    dW1 = np.dot(dh.T, processed_states)
    return {1:dW1, 2:dW2}
```

Policy backwards gives us the updated values for policy. First, we multiply the hidden layer h from above and the action gradient we got above. Then, we calculated the first value by multiplying the reversed hidden layer with the states, returning both gradients for the update.

We also quickly discount the rewards from up to down, applying the discount to every single reward so that exploration is encouraged.

```
def discount_rewards(self, rewards):
        discounted_reward = np.zeros_like(rewards, dtype=
            np.float64)
        total_rewards = 0
        for t in reversed(range(0, len(rewards))):
            total_rewards = total_rewards * self.discount
```

```
                + rewards[t]
            discounted_reward[t] = total_rewards
        return discounted_reward
```

Now, we are on the the training process:

```
def train(self, episodes, batch_size, cmax):
        start_now = datetime.datetime.now()

        for epoch in range(episodes):
            state = self.env.reset()
            prev_state = 0
            sstate, shidden, sgrads, srewards = [], [],
                [], []
            done = False
            counter = 0
            grad_buffer = {k: np.zeros_like(v) for k, v
                in self.model.items()}
            rmsprop_cache = {k: np.zeros_like(v) for k,
                v in self.model.items()}
            reward_sum = 0
```

We start by initializing all our memory data. sstate, shidden, sgrads, srewards, are all saved arrays of previous grads that are used to update gradbuffer which is used to update model. rmspropcache is a model coverger that forces it to move towards the center.

```
            while not done and counter != cmax:
                counter += 1
                append_state = state - prev_state

                aprob, hid = self.policy_forward
                    (append_state)
                if random.random() <= self.exporation_rate:
                    action = np.random.choice
                        (np.arange(self.num_actions))
                else:
                    max_prob = np.max(aprob)
                    second_max_prob = np.partition(aprob,
                        -2)[-2]
                    probability_difference = max_prob -
                        second_max_prob

                    threshold = 0.1

                    if probability_difference > threshold:
                        action = np.random.choice(np.arange
                            (self.num_actions), p=aprob)
```

```
        else:
            action = np.argmax(aprob)
        action = np.random.choice(np.arange
            (self.num_actions), p=aprob)
```

This bit chooses an action using policy forward and softmax. Threshold is a variable that is changable within the policy (I totally should make it a variable that can be initialized, but ;D) that is used to determine if the choice should be random based probability or an immediate max. This is because if the difference between the probabilities is too small, then finding the probability would essentially be random, thus the need for the threshold.

```
        action_one_hot = np.zeros(self.num_actions)
        action_one_hot[action] = 1


        sstate.append(append_state)
        shidden.append(hid)
        sgrads.append(action_one_hot - aprob)

        state, reward, done, _ = self.env.step
            (action)
        srewards.append(reward)

        reward_sum += reward

    vstate = np.vstack(sstate)
    vhidden = np.vstack(shidden)
    vgrads = np.vstack(sgrads)
    vrewards = np.vstack(srewards)


    discounted_vrew = self.discount_rewards
        (vrewards)
    discounted_vrew -= (np.mean(discounted_vrew)).
        astype(np.float64)
    discounted_vrew /= ((np.std(discounted_vrew)).
        astype(np.float64) + 1e-8)

    vgrads *= discounted_vrew
    grad = self.policy_backward(vstate, vhidden,
        vgrads)
```

This part reshapes the inputs and resulting variables so that the variables can be put through the policy backward function to get the gradient. While the reward isn't directly appended, it is multiplied by the vgradients in order to show which states and which actions don't work out that well.

```
          for k in self.model:
              grad_buffer[k] = grad_buffer[k].astype
                  (np.float64)
              grad[k] = grad[k].
                  astype(np.float64)
              grad_buffer[k] += grad[k]


          self.learning_rate = self.init_learning *
              np.sqrt(1 - self.decay ** epoch) / (1 -
                  self.discount ** epoch + 1e-8)
          self.exporation_rate -= self.exporation_rate /
              episodes
          if epoch % batch_size == 0:
              for k,v in self.model.items():
                  g = grad_buffer[k]
                  rmsprop_cache[k] = self.decay *
                      rmsprop_cache[k] + (1 - self.decay)
                          * g**2
                  self.model[k] += (self.learning_rate *
                      g / (np.sqrt(rmsprop_cache[k])
                          + 1e-8))
                  grad_buffer[k] = np.zeros_like(v)
```

This bit of code adds the calculated gradient to gradbuffer, calculates the learning rate with decay over time, and reducing the exploration rate size. Then, using the rmspropcache to converge the gradbuffer for every section of the model.

```
          Util.Util.progress_bar(50 * ((epoch + 1) /
              episodes), 50, 'Epoch', f"Total Time Elapsed:
                    {datetime.datetime.now() - start_now}
                        || Reward: {reward_sum}          ")

          self.plot_for_train.append(epoch)
          self.reward_list.append(reward_sum)

      self.save_model()
      winsound.PlaySound("sgm.wav", winsound.SND_ASYNC |
          winsound.SND_ALIAS )
      plt.plot(np.arange(episodes), self.reward_list)
      plt.show()
```

Finally, this is just some aesthetic stuff, with a progress bar, plotting the data, and of course, playing sgm.wav everything is done to notify people (sgm.wav is Saul Goodman's Theme from Better Call Saul :l)

All in all, we have some basic Reinforcement Learning code from only numpy. (Full code: https://github.com/Xild076/poly-stock-ai/blob/main/Policy.py)

3 Development of the Stock Environment

Now, we got the Stock Environment. First, we start with data collection, the most important step of the process.

```
def get_stock_data(self, keys, start_day, end_day):
        #Getting data based on days

        ticker = yf.Ticker(keys)
        ticker_hist = ticker.history(period='max',
            start=start_day.strftime('%Y-%m-%d'),
                end=end_day.strftime('%Y-%m-%d'),
                    interval='5d')
        avg_total = []
        indexi = ticker_hist.index.values
        time_list = []
        for i in range(len(indexi)):
            time_list.append(Util.dt64todt(indexi[i]))
            avg_total.append(ticker_hist.values[i][3])
        return np.array(time_list), np.array(avg_total)
```

This code, with the module yfinance, pulls data from a certain time period and returns 3 things: the dates of the data and the average value of the training. This is done with the Ticker module and getting the pandas data to average out. This is put into stock data dump.

```
def stock_data_dump(self, keys, start_day, end_day,
    force = False):
        #Dumps stock data

        for key in keys:
            if not force:
                if not Util.file_exists
                    (f'saved_stocks\{key}.npy'):
                    time_list, data_list =
                        self.get_stock_data(key, end_day,
                            start_day)
                    Util.save_load(2, f'saved_stocks\{key}.npy',
                        np.array([time_list, data_list]))
            else:
                time_list, data_list = self.get_stock_data
                    (key, end_day, start_day)
                Util.save_load(2, f'saved_stocks\{key}.npy',
                    np.array([time_list, data_list]))
        print("")
```

This code pulls the data from the internet and saves that data within the system so the training doesn't have to pull data from the internet every single time, which saves quite a lot of time.

```
def get_stock_daycount_data(self, key, day, day_count):
        # Get stock data from file or cache
        if key not in self.dict_of_stock_files:
            all_array = self.load_stock_data(key)
            self.dict_of_stock_files[key] = all_array
        else:
            all_array = self.dict_of_stock_files[key]

        dates, prices = all_array[0], all_array[1]
        nearest_date, index =
            Util.find_nearest(dates, day)
        return dates[index : index + day_count],
            prices[index : index + day_count]
```

The culmination of all this code is getting the data based on a day count, which would provide one with x amount of days of data with a certain start date. This same concept is applied to FRED, or Federal Reserve Economic Data database to get data about the economy, using datareader from pandas. Sometimes, some parts of the datareader produce null values, so we have to replace them with the average in order to make sure every part is filled.

```
def get_fred_data(self, code, start_day, end_day):
        data_source = 'fred'
        data = DataReader(name=code, data_source=data_source,
            start=end_day, end=start_day)
        data_np = np.array(data)
        data_list = []
        mean = 0
        for i in range(len(data_np)):
            if not math.isnan(data_np[i][0]):
                mean += data_np[i][0]
        mean /= len(data_np)
        for i in range(len(data_np)):
            if math.isnan(data_np[i][0]):
                data_np[i][0] = mean
            data_list.append(data_np[i][0])
        return np.array(data.index), np.array(data_list)
```

With the collection of the FRED data, the rest of the procedures included above are implemented and the data is saved and collectable. The last bit of data collection comes with news collection. This is done with requests, pulling the received URLs and pulling the text out.

```
@staticmethod
def fetch_news_from_url(url):
        session = requests.Session()
        try:
            response = session.get(url)
        except:
            return []

        news_articles = []
        if response.status_code == 200:
            soup = BeautifulSoup
                (response.text, features='xml')
            items = soup.find_all('item')

            for item in items:
                title = item.title.text
                description = item.description.text

                news_articles.append({
                    'title': title,
                    'description': description,
                })

        return news_articles
```

This first, all the saved news articles are put into a system where between certain dates, news articles are fetched from Google. By adding the stock symbol on the base of google news, this can pull the first some amount of news articles from the web, putting the text into a list.

```
@staticmethod
def fetch_stock_news_between_dates(stock_symbol,
    start_date, end_date, batch_size=5):
        base_url = 'https://news.google.com/rss/search?'
        news_articles = []

        with ThreadPoolExecutor() as executor:
            futures = []
            current_date = start_date

            while current_date <= end_date:
                date_str = current_date.strftime('%Y-%m-%d')
                params = {
                    'q': f'{stock_symbol} stock news',
                    'hl': 'en-US',
                    'gl': 'US',
                    'ceid': 'US:en',
                    'geo': 'US',
                    'ts': f'{date_str}T00:00:00Z'
                }
                url = base_url + urlencode(params)
                futures.append(executor.submit(BasicSentAnalyzer
                    .fetch_news_from_url, url))
                current_date += timedelta(days=1)

            for future in futures:
                news_articles.extend(future.result()
                    [:batch_size])

        return news_articles
```

After all this is done, all the the news articles are placed into the nltk sentiment analyzer for analysis, which is averaged out.

```
@staticmethod
def calculate_average_sentiment(news_articles):
        total_sentiment = 0
        num_articles = len(news_articles)

        sid = SentimentIntensityAnalyzer()

        for article in news_articles:
            text = article['title'] + '_' + article
                ['description']

            sentiment = sid.polarity_scores(text)
                ['compound']
            total_sentiment += sentiment

        if num_articles > 0:
            average_sentiment = total_sentiment
                / num_articles
            return average_sentiment
        else:
            return 1
```

Now that all the data preparation is finished, the rest is a simple environment formation. A random date between the data range is round, and the data including dates, stock value, FRED dates, FRED data, and new data are all gathered into a state and normalized. The action is calculated from 0 to 2 times the acting detail based on this formula: $mappedvalue = math.floor((percentchange - 1)/(self.upto/self.actdetail)) + self.actdetail + 1$, of which the AI needs to find. All of this comes together to from an environment where the max score is 500 if the AI gets all the guesses correct and -1000 if the AI gets all the guesses wrong. The environment can be placed into the Policy Algorithm for it to train.

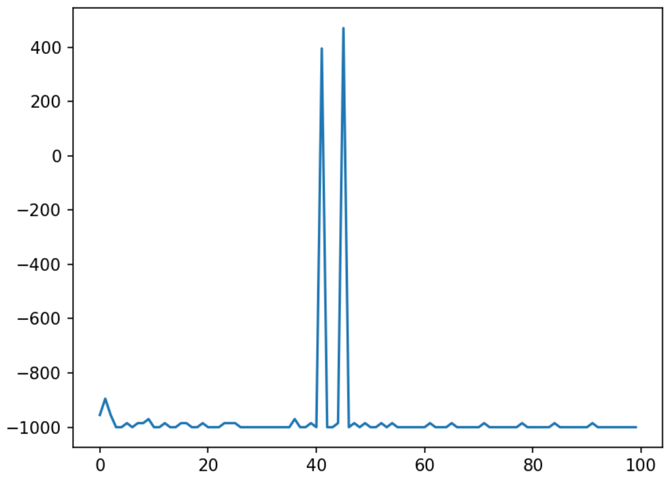

Figure 1: 100 training steps

4 Training and Results

Unfortunately, training is extremely time intensive. Saving the data onto the drive was a method to increase speed, however, the most of the time during training comes from the online search. On average, online search takes up 85 percent of the entire training time. For overall training time, each step seems to take 4.5 seconds on average, so training for 1000 epochs would take about 1 hour and 15 minutes, and training for 10000 epochs would take about 12 and a half hours, which is the maximum I've trained it for. In addition, it relies on constant input of web data, thus if internet connection is cut, though I have a failsafe, it is still problematic.

The results are not too flattering. As seen in the image (figure 1), the AI normally fails to predict anything unless by accident. Based on interpretation, I can say one of three things. First possibility: my algorithm fails, which I'm pretty sure it doesn't because I have trained it with other environments. Second possibility: I haven't trained it enough, and this is a big possibility. As noted above, the time it takes for about 10000 epochs would be about 12.5 hours, which I don't have the resources for. If anyone else would like to train it more, they are free to do so, but this also leads me to the third possibility. Third possibility: The stock market is impossible to predict. This is the biggest possibility. The stock market is volatile, chaotic, and messy. Because of this, predicting the stock market might prove difficult. In addition, one common attribute I noticed is that in the short run, predictions are always one number, more commonly the middle number. However, as training progressed, the number became seemingly random. All this leads me to the conclusion that it seems either the AI will go for the completely safe action of going for the middle or will go for random actions in order to get lucky as to get a higher score by accident.

5 Conclusion

The conclusion is unfortunately that an AI predicting the stock market doesn't seem possible with current tech. Again, as noted above, perhaps better training or things like that can fix such issues. In the future, I'm sure someone will be able to figure something out, but for now, I shall get some sleep knowing I cooked.