Hedging Basis Risk
Using Reinforcement Learning

Samuel Watts

sam@samueldwatts.com

University of Oxford

November 2015

Contents

1 Introduction

This paper investigates the feasibility of using machine learning techniques, in particular reinforcement learning, to hedge basis risk. Basis risk is the risk that the offsetting investment of a hedging strategy does not perfectly hedge due to imperfect correlations, causing excess profits or losses. This paper focuses on the problem discussed in Monyios' lecture notes [4] of hedging a short put position on a non-traded asset by trading a portfolio of Δ_t of a correlated but tradable asset, and cash.

In [2], Monoyios used a distortion method to derive a formula for the claim's asking price and, from this, a formula for the optimal hedging strategy. These formulae however required the knowledge of the assets' drifts. Asset drifts are notoriously difficult to estimate accurately from sample data, as discussed in [3]. Thus for real world applications it may be more effective to consider approaches that do not require knowledge of the assets' drifts. One such approach is reinforcement learning, which instead of calculating an optimal strategy, infers one from learnt experience.

A linear, gradient-descent SARSA(λ) algorithm with binary features and an ϵ-greedy policy is implemented and its performance compared with Monoyios' optimal hedging strategy.

The structure of this paper is as follows: We first introduce reinforcement learning and explain the key concepts and pertinent extensions. Then we describe how a reinforcement learning model can be used to hedge basis risk. In the following section we present the results of an implementation of such a model and compare it with Monoyios' utility-based hedging strategy from [2].

2 Introduction to Reinforcement Learning

This section is a summary and discussion of the pertinent aspects of reinforcement learning from the classic introductory text to reinforcement learning by Sutton and Barto [7].

Reinforcement learning is a well established machine learning technique for generating optimal strategies for an agent interacting with an environment to achieve a goal. Rather than strategies such as supervised learning that require input of labeled examples of optimal behaviour from a knowledgeable supervisor, or unsupervised learning that tries to find hidden structures in unlabeled data, reinforcement learning finds optimal strategies by receiving feedback on its performance directly from the environment. This feedback provides reinforcement of what is a 'good' and what is a 'bad' strategy.

A reinforcement learning system consists of five main sub-elements: an agent, an environment, a policy, a reward signal and a value function. Additionally there can also be a model of the environment.

An *agent* is the decision making object or algorithm. At each time-step it can receive some information on the current state of the *environment* (which is defined as everything outside itself). From this information the agent must decide on an action to take. It decides which action to take based on its *policy*, a mapping from perceived states to actions

to be taken in that state. The action policy completely defines the agent's behaviour. Often the agent's actions affect the environment, creating a feedback loop.

Due to the results of its actions the agent receives a single number, a 'reward', from its environment. This is called the *reward signal*, although it can also be negative, i.e. considered a 'punishment'. The agent's goal is to maximise its reward over the long-run.

Using this feedback on its actions the agent updates its *value function*. This approximately maps each state-action pair to the total rewards in the future it currently expects to accumulate by taking this action in this state. This value function is used to improve the agent's action policy. Since the value function takes into consideration future rewards the action policy is more likely to recommend the best action to take in each situation to achieve the agent's long-term goal, not just maximise rewards in the short-term.

Over time the agent's action policy improves as it receives more and more feedback on its decisions and its value function becomes more accurate. This is an intuitive learning strategy not only for machines but also for animals, children and even video game players. To put it more simply: try something, see whether you get rewarded or punished, remember it, and when faced with that situation again use your previous experience to decide which action to take.

The idealised form of this reinforcement learning task is a Markov Decision Process (MDP). These are discrete time processes where at each time step the environment is in state s and the agent must decide on an action, a. The environment then moves to a new state s' giving the agent a reward r. The MDP is defined by its state-action set and environment dynamics. Theses dynamics are specified by the probability, given state s and action a, of the next state and reward, s' and r:

$$p(s', r|s, a) := \mathbb{P}[S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a]$$

From this we can compute the state-transition probabilities:

$$p(s'|s, a) := \mathbb{P}[S_{t+1} = s' \mid A_t = a] = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

where $\mathcal{R}$ is the reward space.

MDPs are useful for studying optimisation problems, in particular stochastic optimisation control problems. In Monoyios' lectures [5], continuous versions of these problems were tackled by taking a dynamic programming approach. We now establish the discrete version of dynamic programming, following [7].

To decide which action to take the agent uses a value function. This function estimates the expected rewards return for the agent in the given state, or for performing a given action in a given state. Since future rewards depend on future actions the value function depends on the action-policy. The value of a state s under a policy π is:

$$v_\pi(s) := \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right]$$

where $\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable given that the agent follows policy π, the return G_t is some specific function of the reward sequence R_t, and γ is a

discounting factor (e.g. interest rate). This is the expected return when starting in state s and following action policy π thereafter.

Similarly we can define the action-value function for policy π as

$$q_\pi(s,a) := \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

This is the expected return starting in state s, taking action a and following π thereafter.

A fundamental property of value functions is that they satisfy the following recursive relationship:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, S_t = s'\right]\right] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \qquad (1)
\end{aligned}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π. This is the Bellman equation for v_π. It is the discrete version of the Bellman equation from the Stochastic Optimisation lecture notes [5]. It states that the value of the start state must equal the discounted value of the expected next state, plus the reward expected along the way. Following a similar proof to the continuous case in the lecture notes [5] we can prove the existence of an optimal policy π_*. This policy gives us the optimal state-value function

$$v_*(s) := \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}$$

Equivalently we also get from π_* the optimal action-value function

$$q_*(s,a) := \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

We can write q_* in terms of v_* as follows:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

The Bellman equation for v_* is special since it does not refer to any specific policy. It is known as the Bellman optimality equation and is expressed as follows:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]
\end{aligned}$$

This means the value of a state under an optimal policy must equal the expected return of the best action from that state. The equivalent Bellman optimality equation for q_* is:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned}$$

In order to compute v_π for an arbitrary policy π we can use the recursive relationship (1). If the environment's dynamics are completely known then (1) is a system of simultaneous linear equations. Alternatively it can be solved iteratively. To do this, the initial approximation of each state, v_π^0, is set arbitrarily. Then each successive approximation is obtained using (1) as an update rule:

$$v_\pi^{k+1}(s) = \sum_a \pi(a, s) \sum_{s',r'} p(s', r|s, a)[r + \gamma v_\pi^k(s')]$$

This sequence v_π^k converges to v_π as $k \to \infty$.

We could perform this for every policy π and then find the optimal policy. This method however is very time consuming since often the policy space is very large and the number of bad policies is much larger than the number of optimal policies. Instead we can compare deterministic policies using our discovered $v_\pi(s)$ values. At state s we wish to know whether choosing $a \notin \pi(s)$ would be better. We can do this by selecting a in s then following the existing policy π thereafter. If this new policy π' gives $v_{\pi'}(s) > v_\pi(s)$ then π' must be a better policy. In other words

$$q_\pi\left(s, \pi'(s)\right) > v_\pi(s) \implies v_{\pi'}(s) > v_\pi(s)$$

This is the policy improvement theorem. Starting from an arbitrary policy π we compute v_π. From this we yield a better policy π'. We can then compute $v_{\pi'}$ and improve it again to yield an even better policy π''. This is called policy iteration and it is straightforward to prove it must converge to the optimal policy for a finite MDP.

If the rewards are stochastic then finding the optimal strategy is a stochastic optimal control problem and can be solved using dynamic programming. However this relies on knowing the state-transition probabilities. If these are unknown a reinforcement learning approach is a natural alternative.

Unlike other methods which try and solve the Bellman optimality equation or find approximate solutions using complete knowledge of the environment, reinforcement learning finds solutions using actual experienced transitions instead of knowledge of the expected transitions. To do this reinforcement algorithms estimate v_π and q_π from experience. An obvious way to get estimates of $v_\pi(s)$ is by sampling each state-action pair (s, a) many times and taking the average returns. This is the Monte-Carlo method. However these estimates do not build upon the estimate of any other state, so called bootstrapping.

The problem with the Monte-Carlo method is that following a deterministic policy will mean many action-state pairs may never be experienced, since only one of the actions

is performed for each state. Additionally, in most applications the environment will not ensure all state-action pairs can be visited equally so some may be experienced infrequently. Thus their q values will not improve. The problem with this is the q values are used to choose the best action to take in each state.

Thus we need to have our agent *explore* to make sure all actions are selected with non-zero probability. A common way of ensuring this is to use an ϵ-greedy policy. This means that in each state the agent chooses the action with the maximal estimated action value most of the time. However, with probability $\epsilon \in (0, 1)$ they select an action at random. This makes sure the agent does not only ever choose the action with the currently estimated maximal value (which may be incorrect) but also tries other actions to improve their value estimate. On an infinite time-line this policy converges to the maximal action values, as proven in [7].

We now turn to bootstrapping methods, for which the simplest reinforcement learning example is Temporal-Difference learning (TD). This estimates (1) by iteratively updating approximations as follows:

$$v_\pi^{k+1}(S_t) = v_\pi^k(S_t) + \alpha[R_{t+1} + \gamma v_\pi^k(S_{t+1}) - v_\pi^k(S_t)]$$

where α is the step-size parameter. This method samples the expected returns and uses the current estimate, rather than the true v_π, thus it does not require a model of the environment or state transition probabilities. It has been proven to converge to v_π for any fixed policy π and sufficiently small α [7].

For our purposes we wish to learn an action-value function, $q_\pi(s, a)$, for the current policy and all states and actions, rather than the state-value function as above. The corresponding TD algorithm is:

$$q_\pi^{k+1}(S_t, A_t) = q_\pi^k(S_t, A_t) + \alpha[R_{t+1} + \gamma q_\pi^k(S_{t+1}, A_{t+1}) - q_\pi^k(S_t, A_t)]$$

We are now considering transitions from state-action pairs to the next state-action pair. Our sequence of events is $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ which gives the algorithm its name, SARSA. According to [7], SARSA converges to an optimal policy and action-value function if, on an infinite time-scale, all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (the policy of always choosing the maximal valued action).

Often the state-action space considered for the problem is very large, as is the case for the problem discussed in the next section. Therefore we need to consider methods that learn more efficiently. One of the basic mechanisms for doing so are eligibility traces. Intuitively these are temporary records of states visited and actions performed that mark these events as eligible for receiving credit or blame for a TD error. From these traces we can track how recently an action-state pair occurred.

The temporary record associated to each action-state is called its *eligibility trace*. At each time step the eligibility traces of all non-visited action-states decays by $\gamma\lambda$, where γ is the discounting factor as before and λ is the trace-decay parameter. For the action-state visited however the trace is increased (to record the visit). There are different tracing

methods. For our algorithm we used an *accumulating* trace where the trace decays like the others but is also incremented by 1.

$$E_t(s,a) = \gamma\lambda E_{t-1}(s,a) + \mathbb{1}[s = S_t]\mathbb{1}[a = A_t], \quad \forall s, a$$

The TD error is the difference between return received and estimated q value, which for action-state value prediction is:

$$\delta_t = R_{t+1} + \gamma q_t(S_{t+1}, A_{t+1}) - q_t(S_t, A_t)$$

After each time step the recently visited states receive an increase of their q value proportional to their eligibility traces

$$q_{t+1}(s,a) = q_t(s,a) + \alpha\delta_t E_t(s), \quad \forall s, a$$

Using this method speeds up learning since proportional rewards are propagated backwards to the action-states that caused the reward. An algorithm incorporating eligibility traces into the SARSA algorithm is called SARSA(λ).

In the following section we explain how the techniques explained above can be used to tackle a financial stochastic control problem.

3 Hedging Basis Risk

We consider the stochastic control problem of hedging basis risk that Monoyios' investigated in the lectures [4] and in [2]. We have an incomplete market of two assets: a traded stock, S_t, and a non-traded asset, Y_t. These follow geometric Brownian motions:

$$\begin{aligned} dS_t &= S_t(\mu_S dt + \sigma_S dW_t) \\ dY_t &= Y_t(\mu_Y dt + \sigma_Y dW'_t) \end{aligned}$$

The Brownian motions W_t and W'_t are correlated.

$$d[W, W']_t = \rho dt, \quad W' = \rho W + \sqrt{1-\rho^2}W^{\perp}$$

The agent has risk preferences expressed via an exponential utility function:

$$U(x) = -\exp(-\gamma x), \quad \gamma \in (0,1)$$

The agent initially takes a position in a European derivative paying n units of the derivative's payoff $h(Y_T)$ at expiry time T. However they can only trade a dynamic self-financing portfolio consisting of Δ_t shares of the traded asset S_t at time $t \in [0,T]$ with the remainder invested in cash paying interest rate r. Due to the imperfect correlation of the assets this generates basis risk.

The agent's optimisation problem is: starting at time $t \in [0,T]$ with endowment $X_t = x$ and initial non-traded asset price $Y_t = y$, find an optimal trading strategy π_* in the class of admissible strategies $\mathcal{P}$ that maximises their expected utility:

$$F^n(t,x,y) := \sup_{\pi \in \mathcal{P}} \mathbb{E}_{t,x,y} U(X_T + nh(Y_T))$$

Following the notes [4] and [2], we will also consider the case of $n = -1$, i.e. a short put position. In [2], Monoyios used a distortion method to derive the following formula for the claim's asking price:

$$p^{ask}(t,y) = \frac{e^{-r(T-t)}}{\gamma(1-\rho^2)} \log\left(\mathbb{E}^0_{t,y}[e^{\gamma(1-\rho^2)h(Y_T)}]\right)$$

From this he derived a hedging strategy for the sale of the claim at the asking price $p^{ask}(t,y)$, which is to hold Δ^{ask}_u shares of the traded asset S at time $u > t$ given by

$$\Delta^{ask}_u = \frac{\rho\sigma_Y Y_u}{\sigma_S}\frac{\partial p^{ask}}{\partial y}(u, Y_u), \quad t < u < T$$

Using a perturbation expansion he gives an explicit result for $p^{ask}_y(t,y)$. However this result requires the input of the two assets' drifts, μ_S and μ_Y. As has been noted in [3], it is very difficult to get a reliable estimate for an asset drift without data spanning hundreds of years. It was also demonstrated that erroneous estimates could often be very destructive. To deal with this risk we take a different approach.

As discussed in the previous section, reinforcement learning can be used to find optimal strategies for stochastic control problems, such as this, without input on the model of the environment. Instead the agent infers the optimal strategy from experience. Thus we can use a reinforcement model to approximate an optimal hedging strategy without requiring a model of the assets. In addition since drifts and volatilities are not constant, a reinforcement learning model can adapt, adjusting its value function to the new environment. Adaptation does however rely on the parameters not changing too quickly for the model to keep up.

In order to analyse this problem, the following reinforcement learning model was used, with the market described above as the agent's environment. We considered the three dimensional state space of: the underlying asset price, the current wealth, and the time (equivalently the time to payoff), (Y_t, X_t, t).

At each time-step based on the current state (y, x, t) the agent must use its policy to decide on an action to take. The possible actions are the same for every state. They are to either: increase, leave unchanged or decrease the agent's current Δ_t by a fixed amount. Δ_t was not allowed to go above 0. If $\Delta_{t-1} = 0$ then choosing to increase it had no effect. This action space is a reasonable proxy for approximating Δ_t and in turn the portfolio distribution. At a cost of preventing the model changing Δ_t by a large amount in a single time-step it reduces the action space, considerably speeding up learning. This action space approach is also used in [1], and seems preferable to the only increase or decrease action space of [6] and others. This is due to the observed behaviour of Δ_t under Monoyios' analytic solution from [2].

After each intermediary time-step the agent receives a reward of zero. At the expiry T however it receives a reward of $U(X_t - \max(K - Y_T, 0))$, in other words the utility of the terminal strategy payoff.

The agent's goal is to find an optimal strategy for modifying Δ_t in each state in order to maximise utility of the terminal payoff.

Each run from time 0 to T is an episode. Many episodes are run in order for the agent to experience a large enough number of action-state pairs to generate a reasonable action-state value function.

Since the state space is not discrete we used generalisation methods to approximate the continuous two-dimensional asset-price space. In particular we used a linear, gradient-descent function approximation for the value function. To discretise the state space into binary features we used tile coding. For details on these methods please see chapter 9 of Sutton and Barto [7]. We omit them for brevity.

The algorithm used is described in Figure 1 below. It was based on an algorithm from [7].

Let θ and e be vectors with one component for each possible feature.
Let $\mathcal{F}_a$, for every action a, be a set of feature indices, initially empty.
Initialise θ with optimistic values, $\theta = 0.03$
Repeat for each episode:
    $e = 0$
    S, A = initial state and action chosen by ϵ-greedy policy
    $\mathcal{F}_a$ = set of features present in S, A (found by tile coding)
    Repeat for each step of episode from 1 to 20:
        For all $i \in \mathcal{F}_a$:
            $e_i = e_i + 1$ (accumulating trace)
        Take action A, observe reward R and next state S'
        $\delta = R - \sum_{i \in \mathcal{F}_a} \theta_i$
        For all $a \in \mathcal{A}(S')$:
            $\mathcal{F}_a$ = set of features present in S', a
            $q_a = \sum_{i \in \mathcal{F}_a} \theta_i$
      A' = new action in S' (chosen by ϵ-greedy)
      $\delta = \delta + \gamma q_{A'}$
      $\theta = \theta + \alpha \delta e$
      $e = \gamma \lambda e$
      $S = S'$
      $A = A'$

Figure 1: Algorithm for linear, gradient-descent SARSA(λ) with binary features, ϵ-greedy policy and accumulating traces.

4 Results

Following Monoyios' implementation in [2] we used the parameters in Table 1 to simulate the basis risk model and his optimal hedging strategy:

Table 1: Model parameters

S_0	Y_0	X_0	K	ρ	r (%)	μ_S (%)	σ_S (%)	μ_Y (%)	σ_Y (%)	γ	Δ_0^{RL}
100	100	8.4806	100	0.75	5	10	25	12	30	0.001	-0.4

The values for X_0 and Δ_0^{RL} input into the reinforcement learning algorithm were taken from the analytic solution for these parameters and the latter rounded. At each time-step the model could choose to increase, keep the same or decrease Δ_t^{RL} by 0.1. This value was chosen by observing the magnitude of changes in the analytic Δ_t.
The model was run for 20 time-steps. This is less than would be desired but was necessary to reduce computation time. To discretise the continuous space into features the tiling method from [7] was used with an 8x8 grid for each time step and two tilings. It was found that increasing tilings any more significantly slowed computation in this case. An $\epsilon = 0.05$ was chosen, as it is a common choice in the literature. The q values were initially all set with an optimistic value of 0.03 in order to encourage more exploratory behaviour initially. This value was found by finding the maximum utility of the optimal hedge strategy and adding a small increment. We set the trace-decay parameter $\lambda = 0.9$ and learning parameter $\alpha = 0.25 = 0.5/m$ as recommended in [7], where m is the number of tilings; in our case 2.
An example episode can be seen in Figure 2. The upper graph shows the traded (blue) and non-traded (red) asset prices over twenty time-steps. The middle graph show Δ_t (blue) and Δ_t^{RL} (red) of the analytic formula and the choices of the reinforcement learning agent. The lower graph shows the resultant wealth process from the two hedging strategies and the terminal payoff, marked with an 'x'.
Figure 3 contains histograms displaying the distribution of the terminal hedging errors $(X_T - h(Y_T))$ for the optimal hedging strategy (upper) and learnt strategy (lower). They are plotted to the same scale for ease of comparison. The resultant statistics can be found in table 2.

Table 2: Hedging error statistics for histograms in Figure 3

	Max	Min	Mean	SD	Median
Optimal hedge	13.8538	-8.0045	5.8651	2.5885	6.4176
Learnt hedge	17.2665	-11.6578	5.8818	3.2394	6.5057

From the histogram and statistics we can see that there is not a significant difference in average performance between the two strategies. The mean and median hedging error were very close, although the learnt strategy was only marginally higher than the optimal strategy. The difference was not signficant.
The learnt strategy had a 25% larger standard deviation in performance however, which could potentially become much more significant over a longer time period.

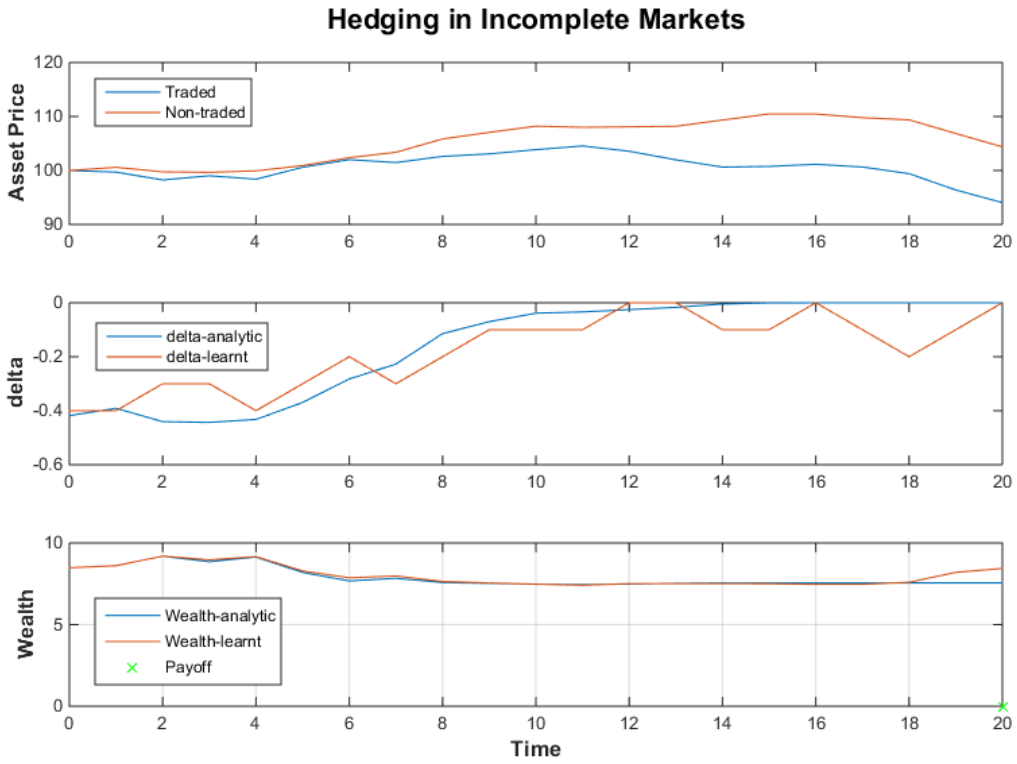

Figure 2: Asset prices (upper), hedge ratios (middle) and hedged portfolio wealths (lower) along a simulated path.

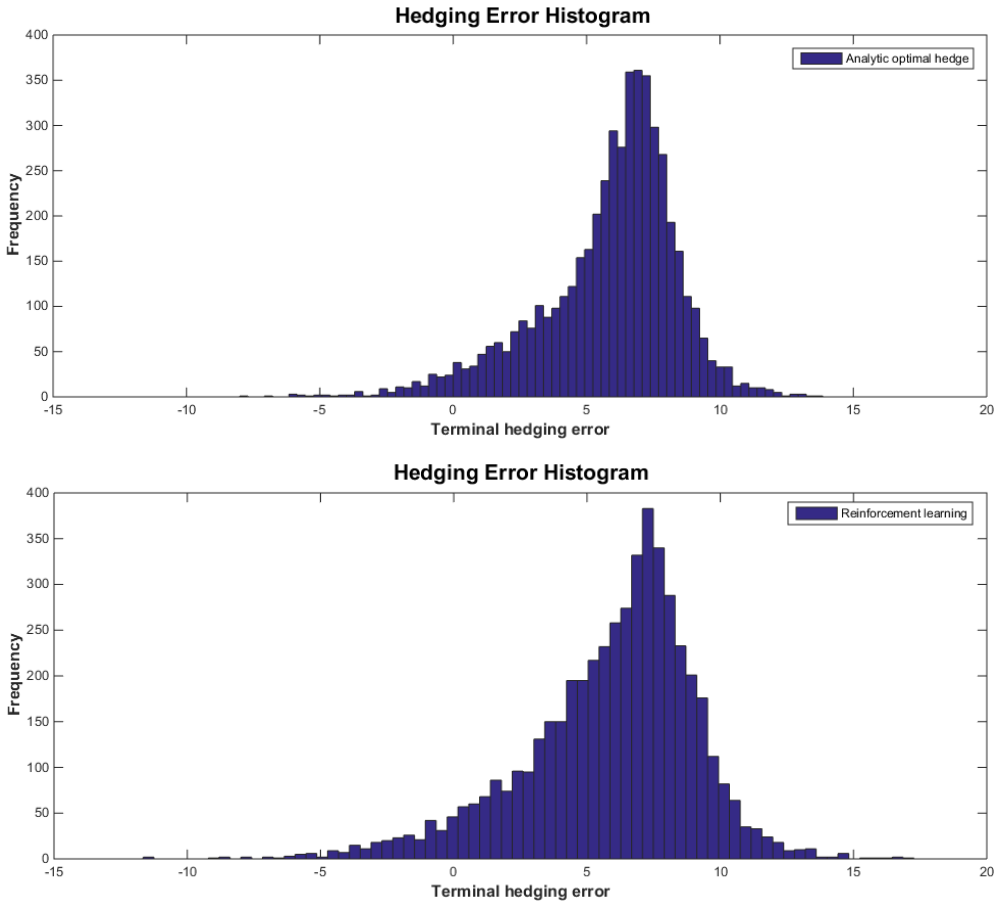

Figure 3: Histograms of terminal hedging error over 5000 sample paths for the analytic optimal hedging strategy (upper) and 5000 episodes for the reinforcement learning algorithm (lower).

5 Conclusions

In this paper it was demonstrated that reinforcement learning is a promising approach in finding a basis risk hedging strategy, especially when the asset dynamics are unknown. From a preliminary investigation it does not appear to perform on average significantly worse than the optimal analytic strategy, although it does have larger variation.
There is plenty of scope for improving this reinforcement learning model. Firstly, additional simple experimentation in varying agent model parameters is required, in order to see if the learning speed and accuracy can be improved. Similarly other variations of reinforcement learning algorithms could be investigated to see if these are more suitable for this task.
The next step would be to perform a real-world implementation of the model using historic rather than simulated data and see if it performs better than the analytic formula using parameter estimates. In order for the model to be practical for day-to-day use however the computation speed would have to be significantly improved. The current implementation takes 9 hours.
It would also be interesting to incorporate transaction costs into the model. Conveniently for reinforcement learning, this would be very straight forward to add to the algorithm.

References

[1] Marco Corazza and Francesco Bertoluzzo. Q-Learning-Based Financial Trading Systems with Applications. *Social Science Research Network Working Paper Series*, October 2014.

[2] Michael Monoyios. Performance of utility-based strategies for hedging basis risk. *Quantitative Finance*, 4(3):245–255, 2004.

[3] Michael Monoyios. Optimal hedging and parameter uncertainty. *IMA Journal of Management Mathematics*, 18(4):331–351, October 2007.

[4] Michael Monoyios. European options in incomplete markets. *MSc Mathematical Finance Lectures, Module 5*, 2015.

[5] Michael Monoyios. Stochastic optimisation. *MSc Mathematical Finance Lectures, Module 3*, 2015.

[6] John Moody and Matthew Saffell. Learning to trade via direct reinforcement. *Neural Networks, IEEE Transactions on*, 12(4):875–889, July 2001.

[7] Richard S. Sutton and Andrew G. Barto. Reinforcement learning an introduction, 1998.