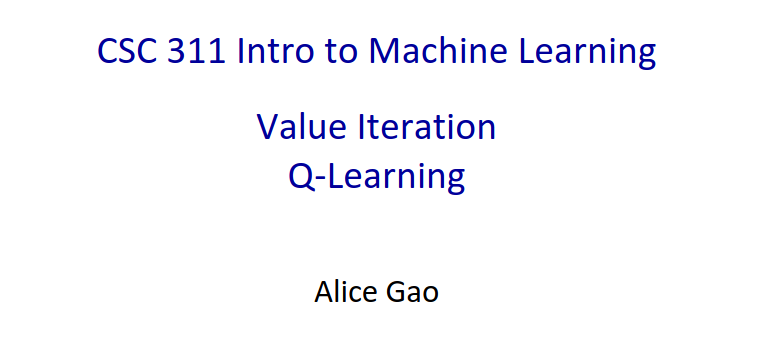
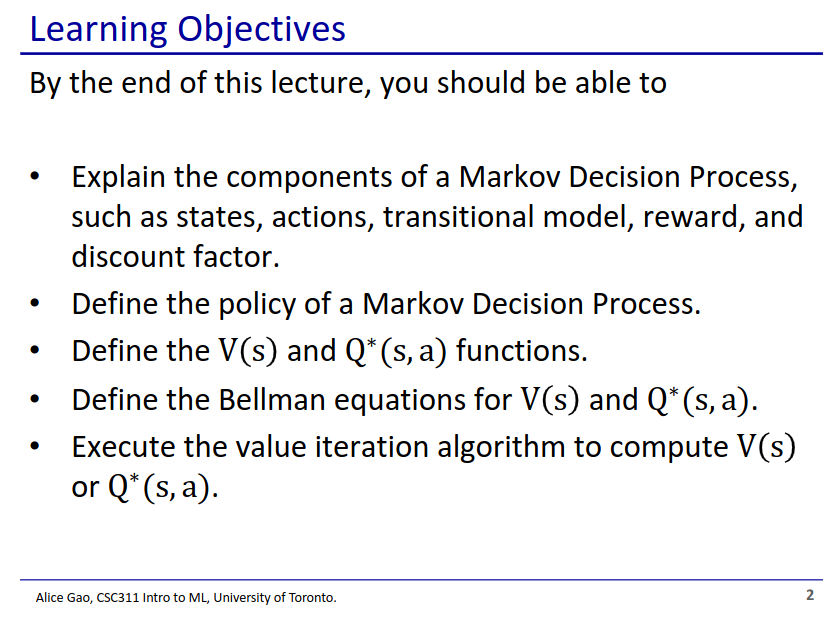
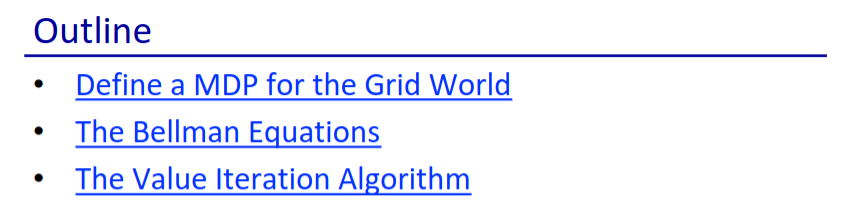
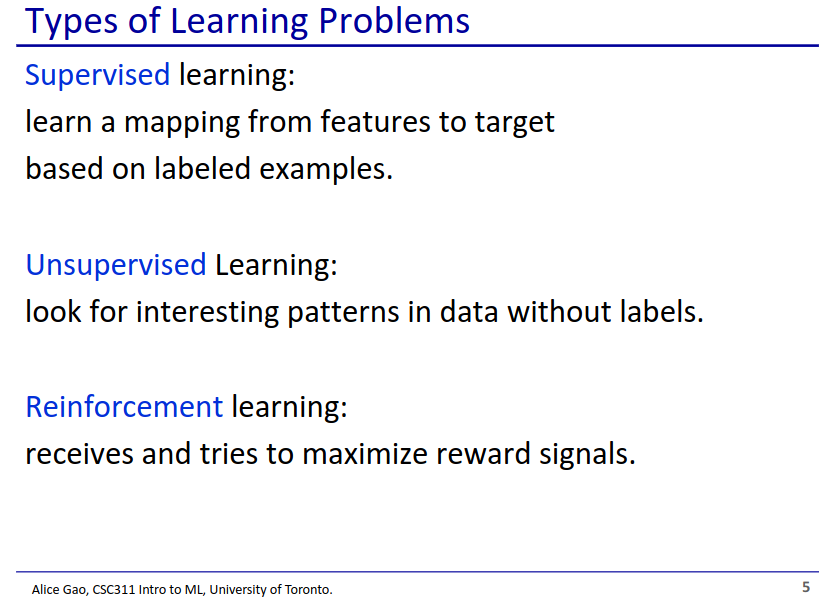
| **Reinforcement learning**   * Model periodically receives reward signals as feedback, and tries to maximise reward   + A type of machine learning technique that is somewhere in-between supervised and unsupervised learning * Model has a notion of “time” such as steps or moves   + Reward is often delayed until after move is made   + Model tries to come up with a sequence of moves that maximises reward   **Grid world example:**   * **Rules**   + Model starts at a cell, and moves step-by-step until it reaches an exit cell   + 2 exit cells, one with a +1 reward and one with a -1 reward   + Model can move in the 4 cardinal directions     - When the model moves, it has a 10% chance to instead move 90 degrees right or left of its intended move * **Model**   + Has 1 state for each cell   + Has 4 actions corresponding to each direction     - Stays in the same cell if it bumps into a wall   + **Reward function**      - The reward a model gets for starting at state s’ and taking action a to end up in state s       * Reward for moving to an exit cell is either +1 or -1 depending on the exit       * We can also apply a small positive or negative reward to landing on other cells     - We can apply a **discount factor** () to future rewards       * Makes immediate rewards more enticing to the model than future rewards   + **Markov assumption**     - The future is independent of the past given the present     - Thus our model's policy only needs to consider its current state/position on the grid * **Optimal policies on the grid world**   + The optimal policy changes based on the reward function   + When the reward for landing on other cells is positive     - Agent will meander around for as long as possible and avoid hitting both exits     - Agent will meander around since stepping on any non-exit cell gives a positive reward   + When the reward for landing on other cells is slightly negative     - Model will try to reach the +1 exit and avoid reaching the -1 exit accidentally     - Model will minimise risks   + When the reward for landing on other cells is very negative     - Optimal policy will be to head towards the nearest exit       * Model doesn’t care which exit it takes, since not taking an exit would cost more   **Bellman equations**   * Equations that equate the value function (V\*) and the Q-function (Q\*) of a reinforcement learning model   + Can be used to determine the optimal policy at each state (\*(s)) * **Value function V\*(s)**   + Expected utility of state s if the agent enters state s and then follows the optimal policy   + V\*(s) = max Q\*(s, a)   + **Bellman equation for V\*(s)** * **Q-function/state-action value function Q\*(s, a)**   + Agent’s expected utility of taking action a while being in state s   + Q\*(s, a) V\*(s)]     - State-action value is the weighted sum of reward and value of all possible resulting states of the action   + **Bellman equation for Q\*(s, a)** * **Finding the optimal policy at state s**   + Optimal action is the one that gives the most utility   + If we know Q\*(s, a):     - \*(s)Q\*(s, a)   + If we know V\*(s):     - \*(s)V\*(s)]       * This is the equation for Q\*(s, a) but substituted   **Value iteration algorithm**   * Bellman equations are nonlinear since they use the max function   + Thus there is not an efficient way to solve them directly   + Instead we can solve them iteratively using the **value iteration algorithm** * Value iteration algorithm works by iteratively getting the value of a state closer to its optimal value |
| --- |



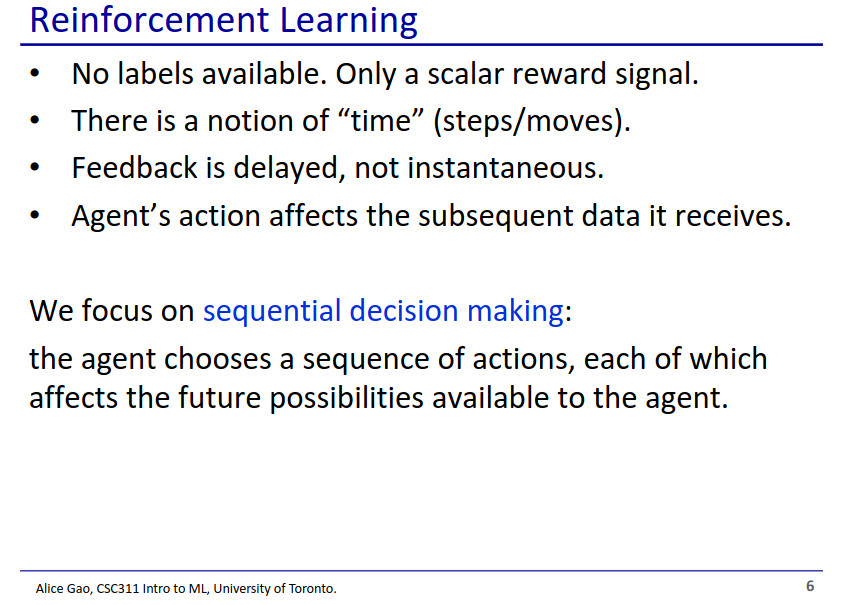




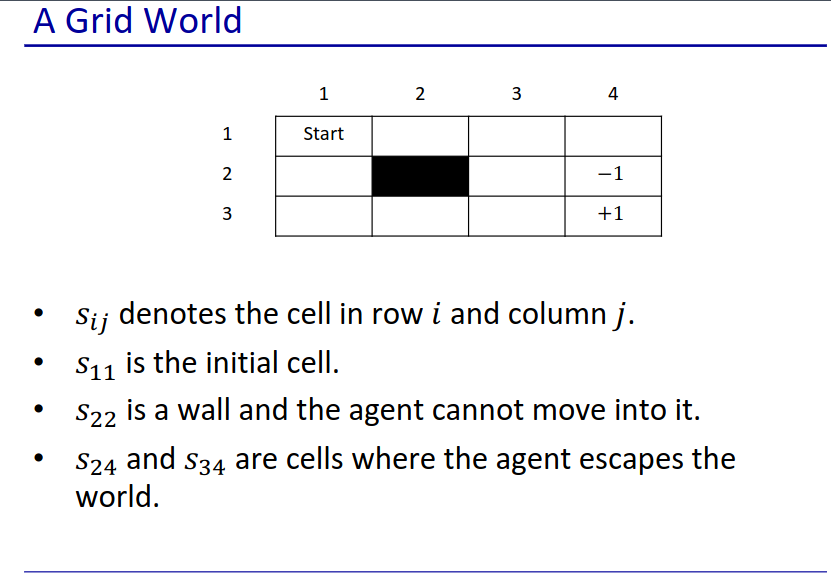




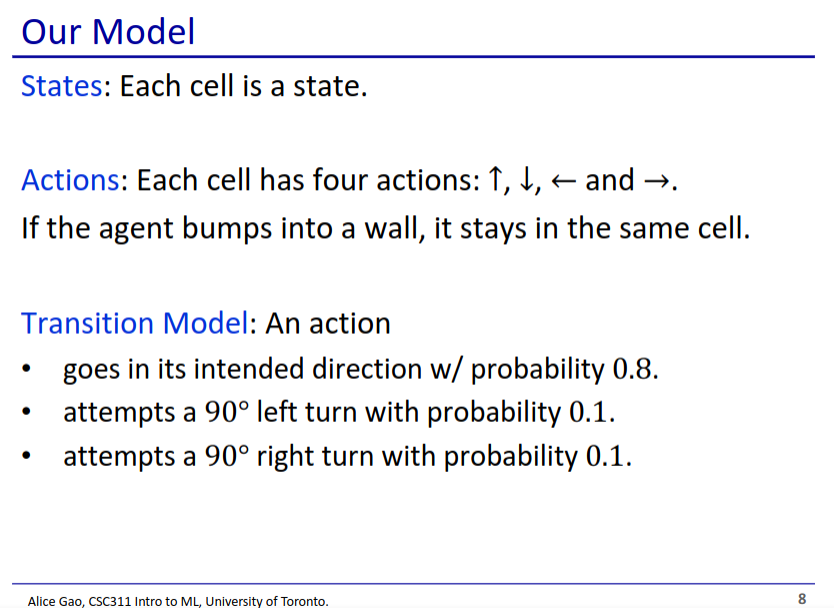
* These are the 3 types of learning problems in machine learning
  + Supervised learning - uses labelled data representing the “correct answer”
  + Unsupervised learning - looks for patterns in unlabelled data
    - We’re just looking for interesting patterns
  + Reinforcement learning - somewhere in the middle
    - We don’t have labels giving answers for everything, but we do have a reward signal that occasionally gives feedback
    - Reward signal is not given at all times, and is often delayed



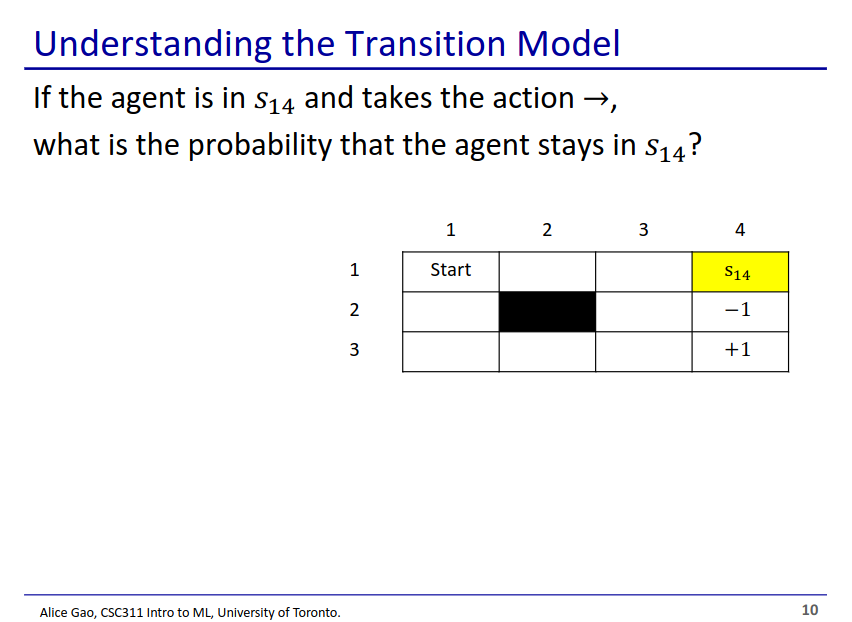
* It’s like playing a game
  + It's not until the end of the game that you know how well you did
* Sequential decision making
  + You choose a sequence of actions
  + Actions affect the future actions you can take
  + An example of a problem like this would be a turn-based game



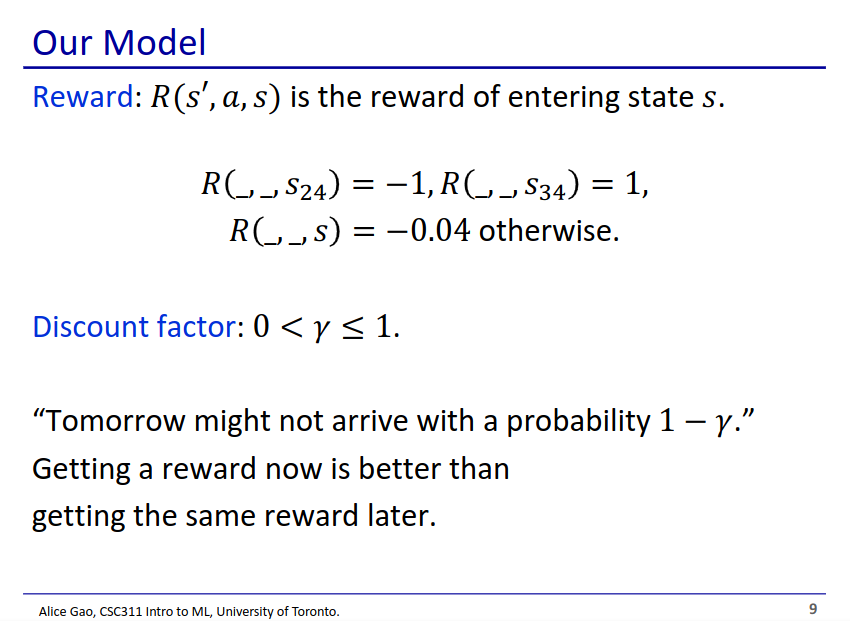
* We always start at the top and try to move to one of the escape squares
  + Ideally we want to move to the positive escape square



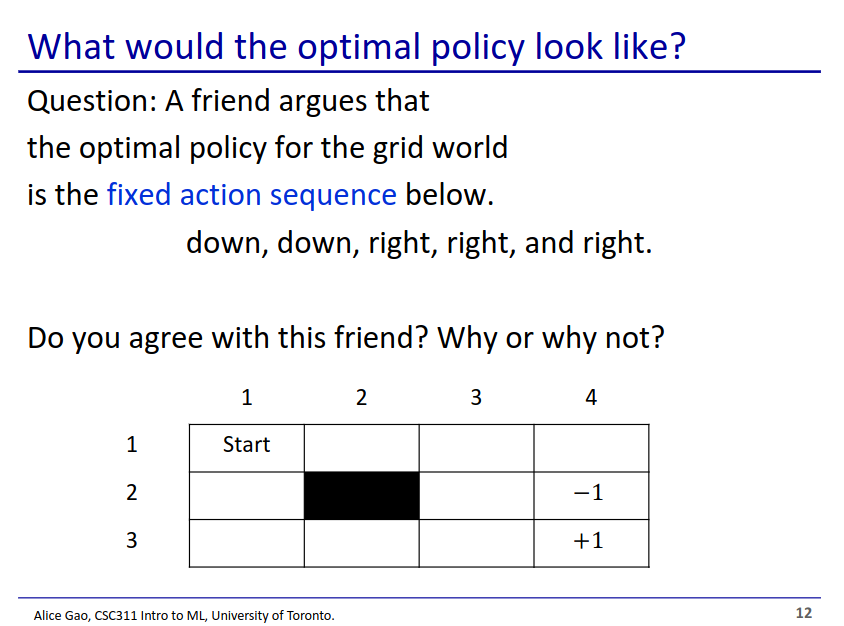
* We want to model our game world as a markov decision process
* Each cell in the grid world that we can stand on has a corresponding state
* Actions: agents can move in any direction
* Transition model:
  + Models that the agent is not perfect
  + 80% of the time it goes in the intended direction
  + 10% of the time it goes 90 degrees to the left of its intended direction
  + 10% of the time it goes 90 degrees to the right of its intended direction



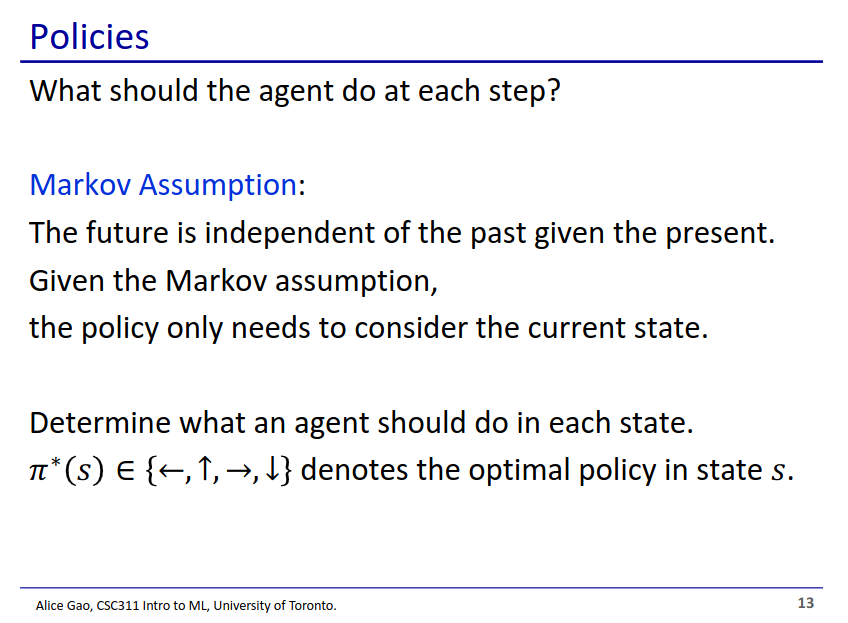
* To the right and up is a wall
  + Going right (80%) results in hitting the wall and staying at s1 4
  + Going up (10%) results in hitting another wall and staying at s1 4
  + Going down (10%) results in moving down into the empty space
* Thus 90% chance of staying

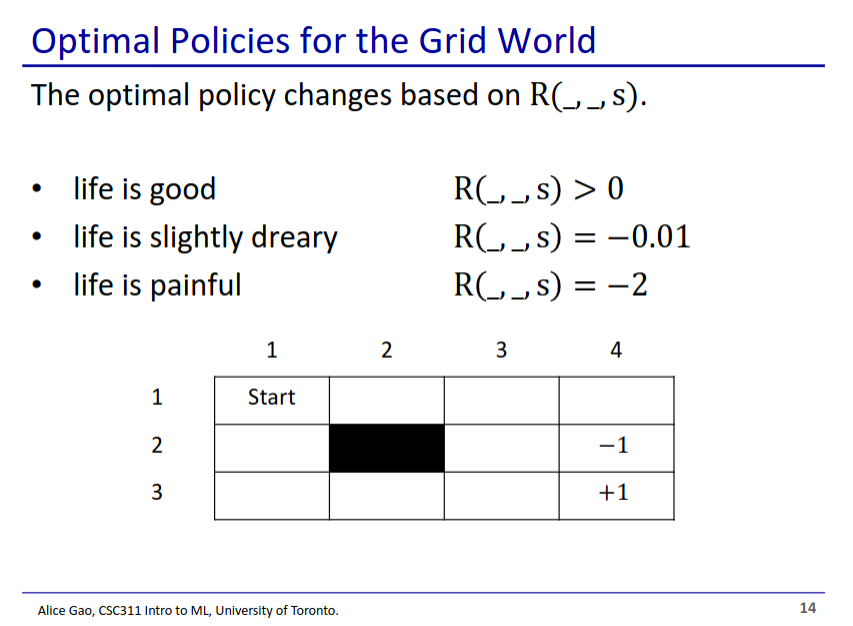


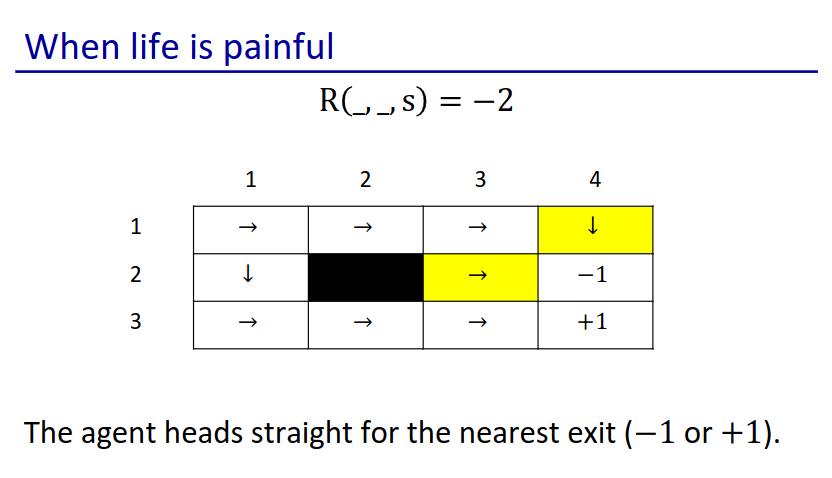
* Reward:
  + The reward we get if we start at state and take action a to end up at state s
* Our reward structure
  + If we end up on s2 4, we get a reward of -1
  + If we end up on s3 4, we get a reward of 1
  + If we end up on any other square, we get a reward of -0.04
* Discount factor
  + Modelling the probability that tomorrow will come
    - Thus the probability that tomorrow will not come is 1-
  + Makes the model prefer a reward now than the reward later
    - Scales down future rewards compared to current rewards
  + Reward sum is thus:



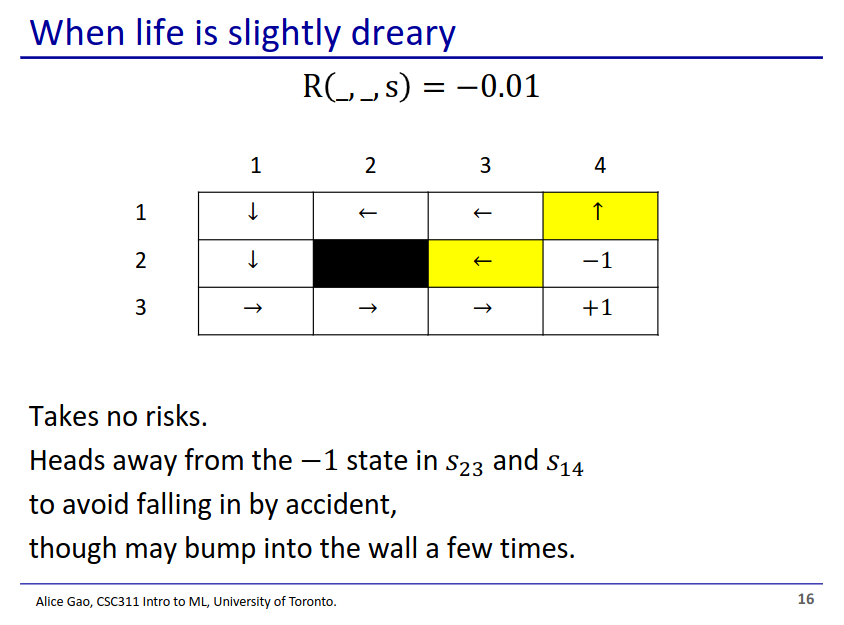
* The transition is uncertain - if we follow the fixed sequence we are not guaranteed to make it to the right square
  + Thus it might be a good idea to not have a fixed action sequence
* The optimum policy will not be a fixed action sequence since it is possible that we will get derailed
  + The optimum policy will have to specify where to go for each possible state - like a contingency plan for if we end up off-course



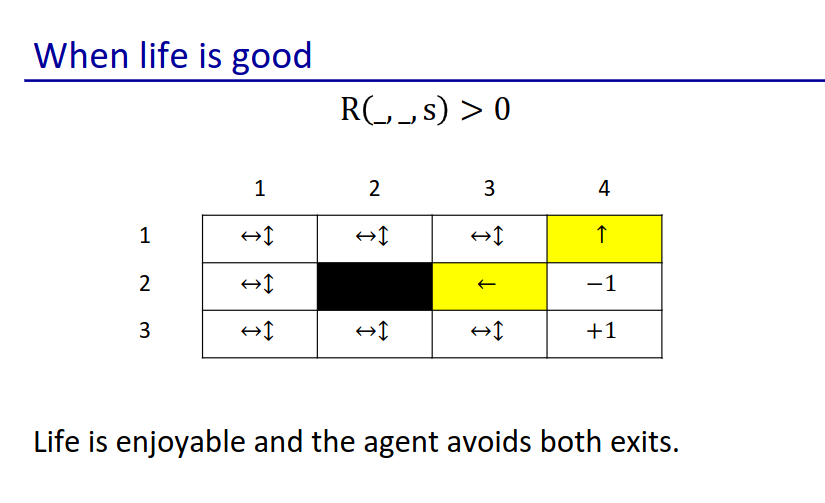




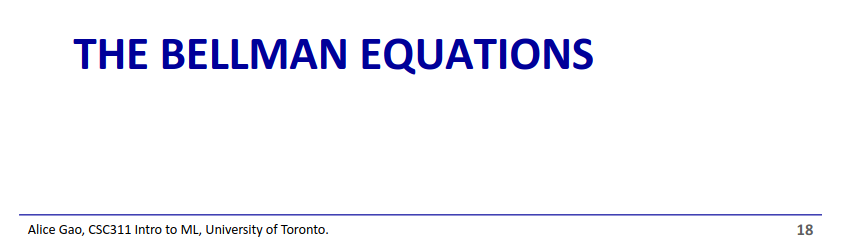
* In this case, our reward function penalises 2 for each step we take
* Life is painful, the way to minimise negative reward is to exit as soon as possible
  + Agent doesn’t really care which exit, just that it can exit
* Me too agent, me too

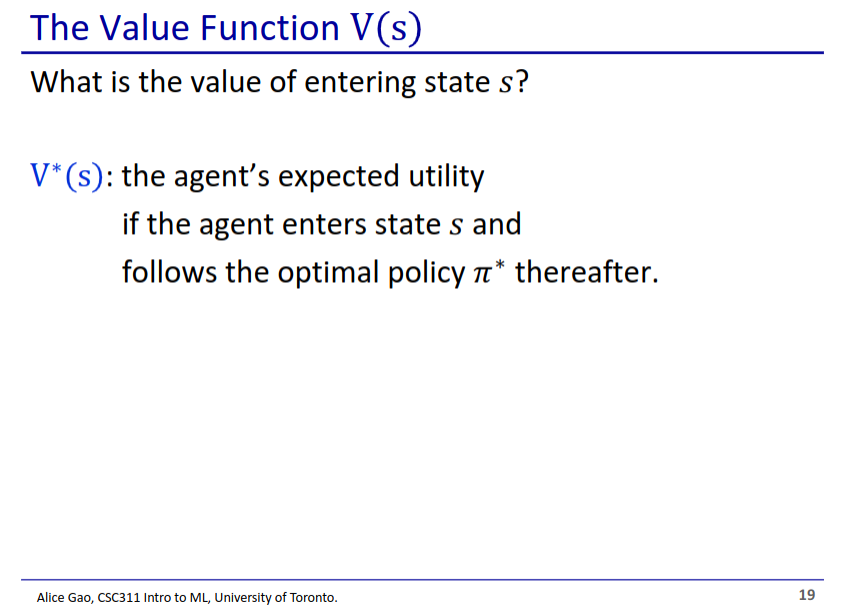


* Reward in this case does not very negatively penalise taking steps
* Agent will minimise risks and avoid hitting the -1 exit
* Agent will meander a little bit before hitting the +1 exit

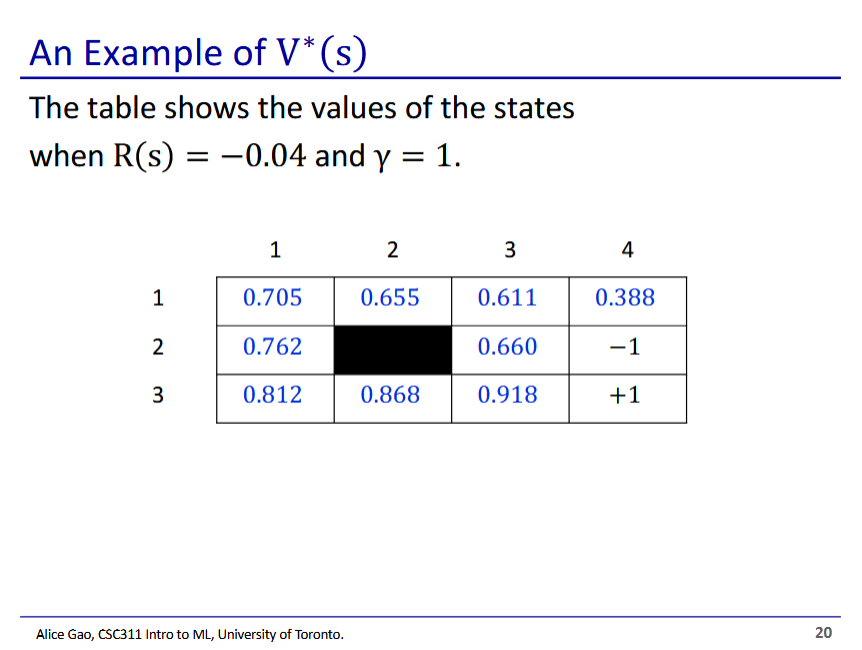


* Life is good: every step gives a positive reward
* Agent will meander around for as long as possible and avoid hitting both exits so that it can gather as much reward as possible

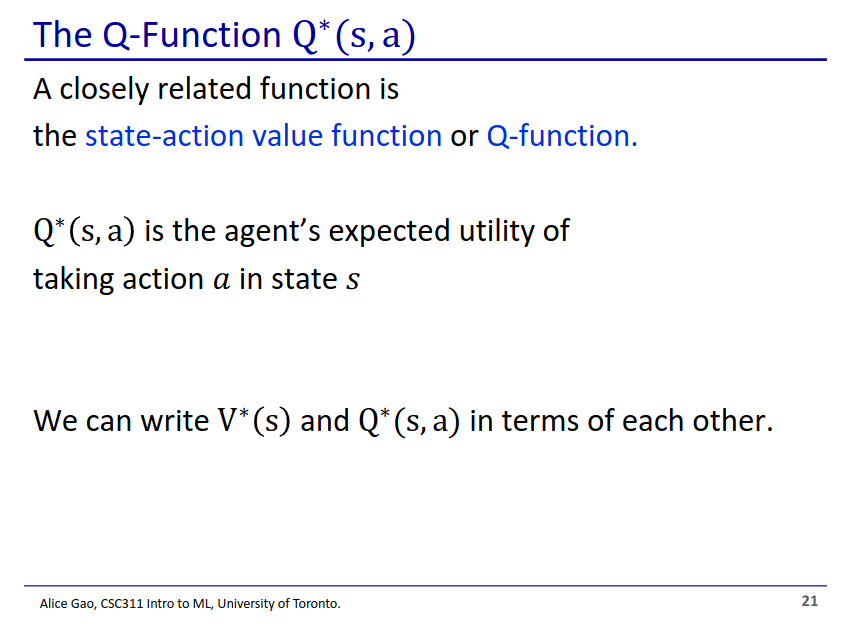




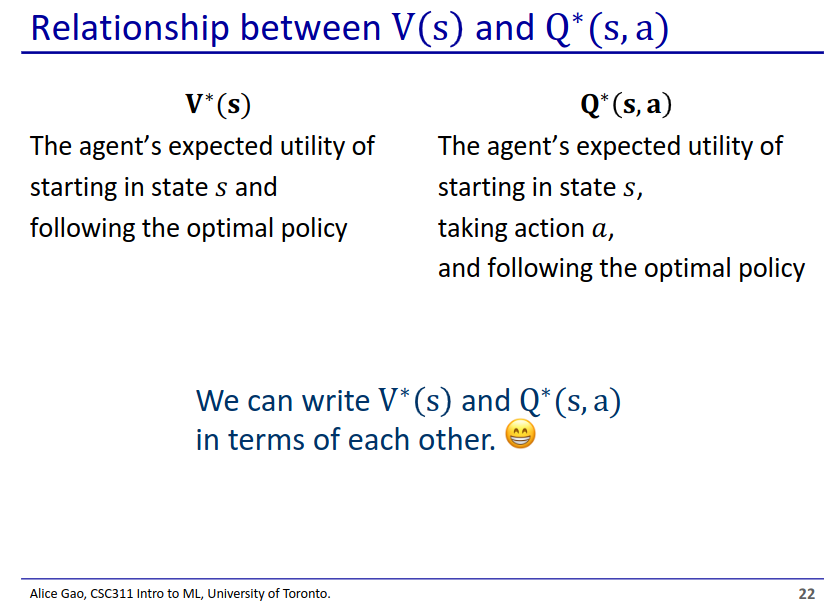
* V\* is very complicated if you actually write it down, so we just abstract it
* Basically is the score the agent can get if it enters s and then follows the optimal policy afterwards

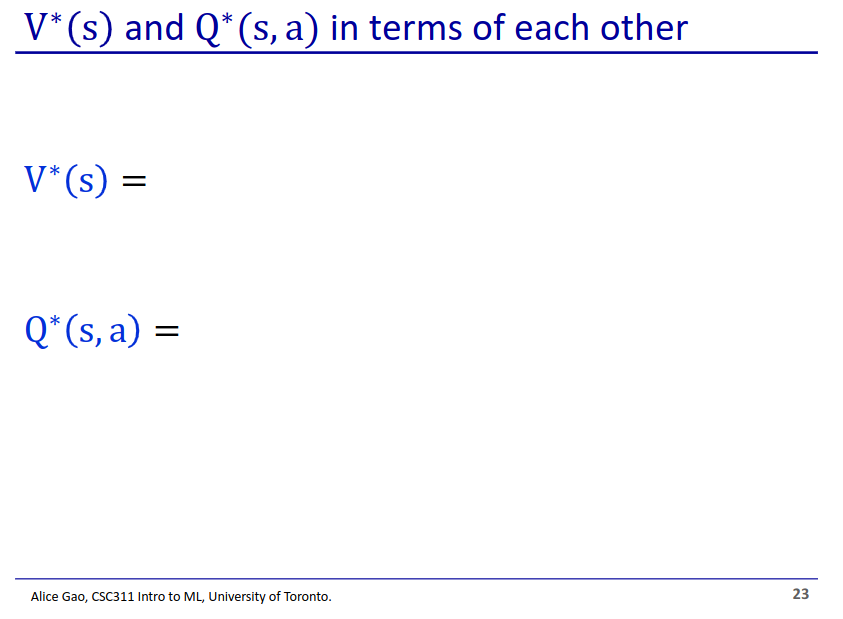


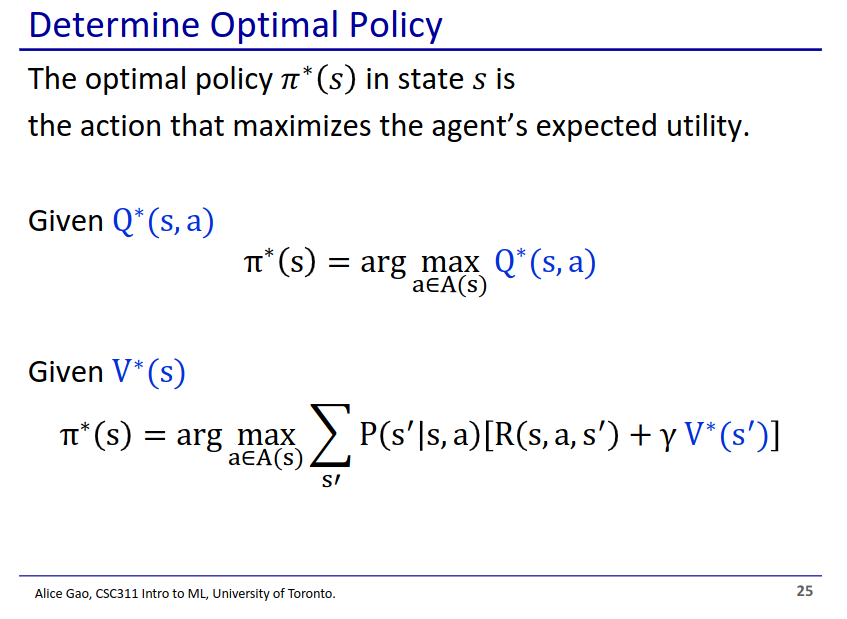
* Our goal is to derive a V\* for every state, and then use that to create the optimal policy
* Value increases as we get closer to the +1 exit, and while we are on the optimal path

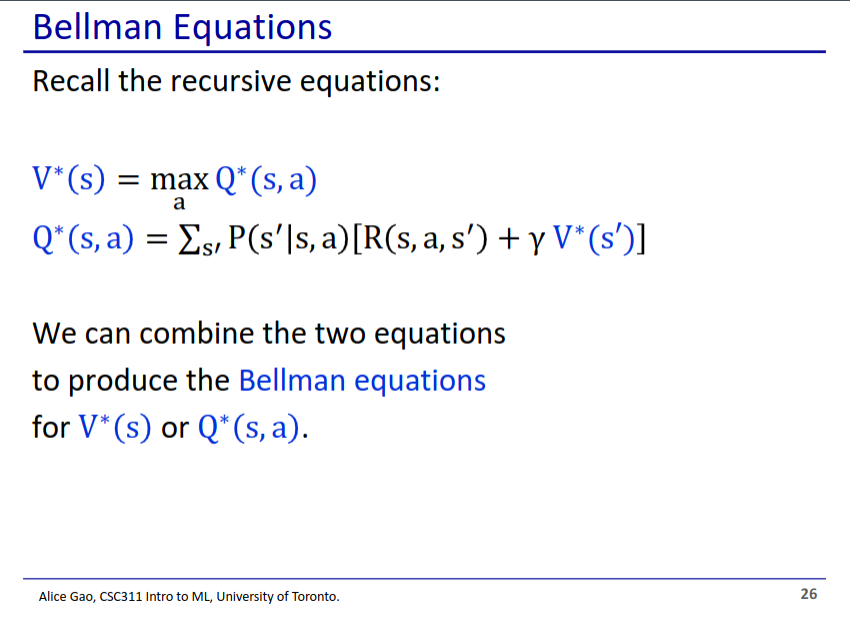


* Q\* is similar to V\*
  + But Q\* takes both a state and an action
* Q\* is the score the agent can expect if it takes action a in state s

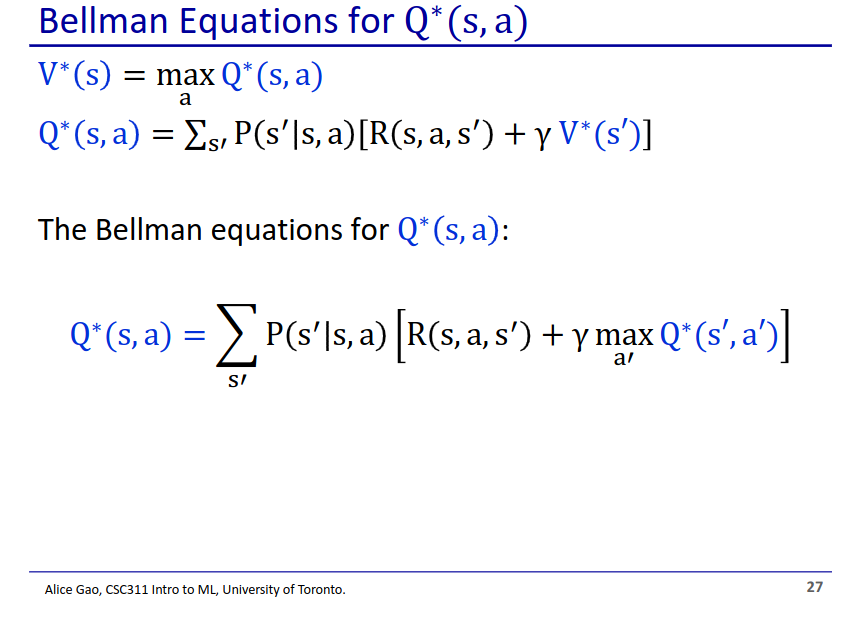




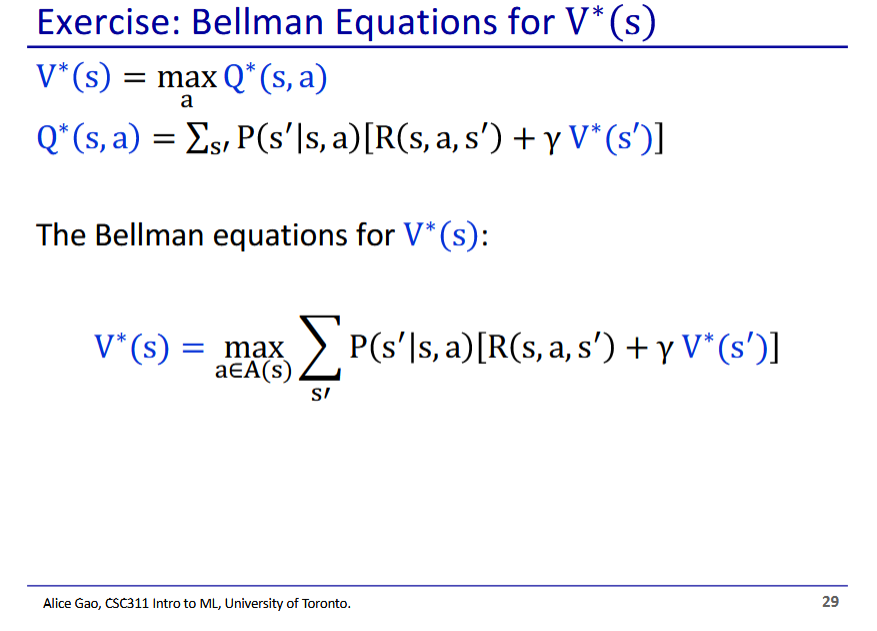




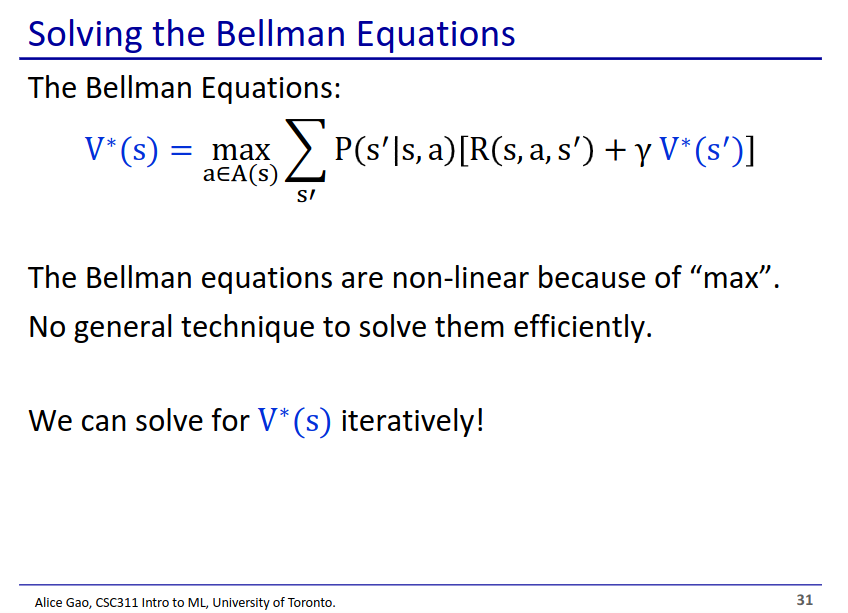
* We solve for the optimum policy by combining the two equations and keeping either just the Q\* or just the V\*

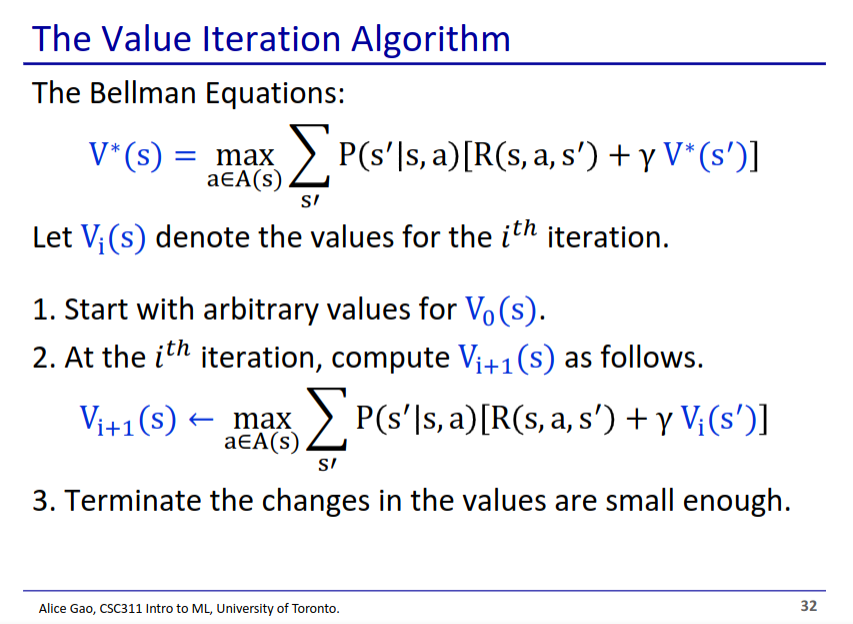


* Substitute in the equation for V\*



* Substitute in the equation for Q\*





* We turn the Bellman equation into an update rule