An Introduction to Reinforcement Learning

Snake Your Way Out of a Paper Bag

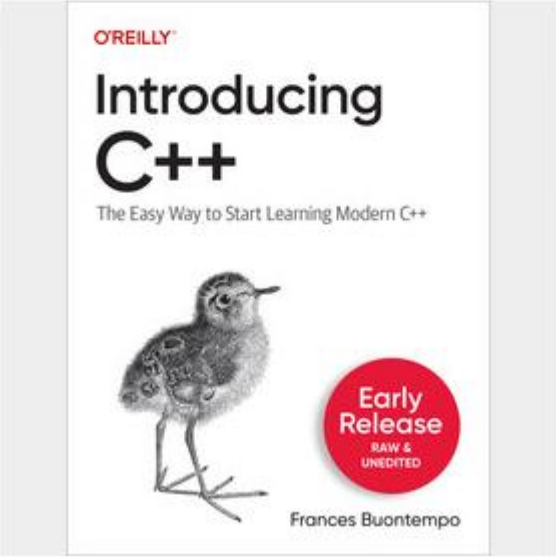
Frances Buontempo

2025

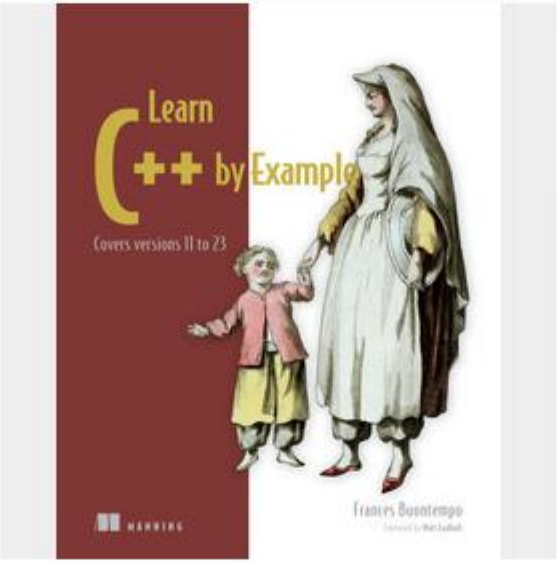
Talk outline

- Overview of reinforcement learning
  - It’s a huge topic so just some parts
- Start simple
  - Move in 1D
- A bit more complicated
  - Move in 2D
- Fun and games
  - Snake!!!!!!!
  - And other arcade games
- (There will be some C++)

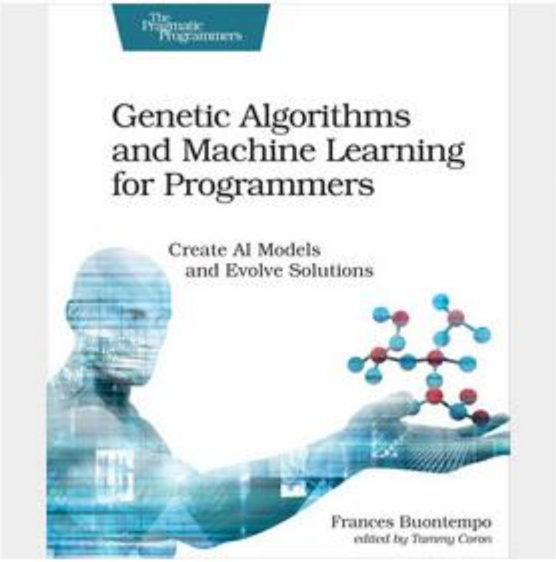
- I edit ACCU's Overload magazine
  - https://accu.org/journals/nonmembers/overload_cover_members/
- Programmer (C++ plus some Python and C#) (mostly in finance)
- Author



Introducing C++



Learn C++ by Example



Genetic Algorithms and Machine Learning for Programmers

Where am I?

- https://mastodon.social/@fbuontempo
- https://bsky.app/profile/fbuontempo.bsky.social
- https://x.com/fbuontempo
    - (formerly https://twitter.com/fbuontempo)
- https://www.linkedin.com/in/francesbuontempo/
- https://buontempoconsulting.blogspot.com/
    - (Sometimes)

Reinforcement learning (RL)

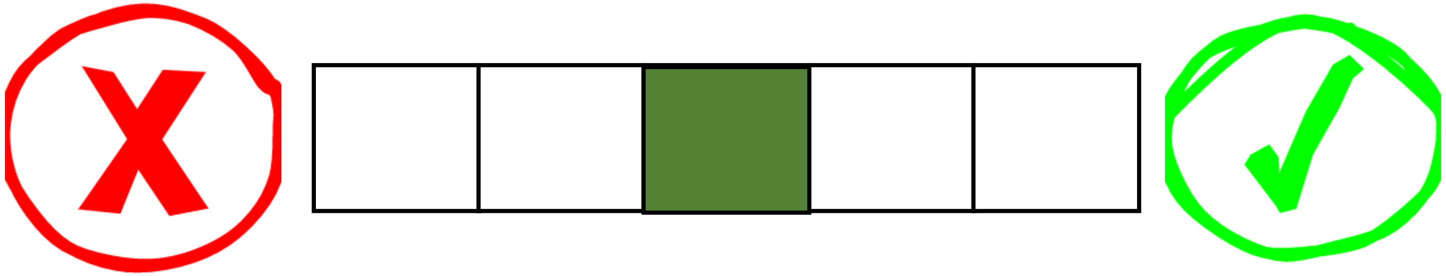
- A type of machine learning, so part of AI.
- Involves agent(s) acting in an environment
  - Static or dynamic
  - Continuous or discrete
  - Action + reward -> learn
- Nothing to do with ChatGPT
  - But Reinforcement Learning from Human Feedback (**RLHF**)
    - Uses human feedback to fine tune LLMs (supervised fine-tuning).

A simple game:

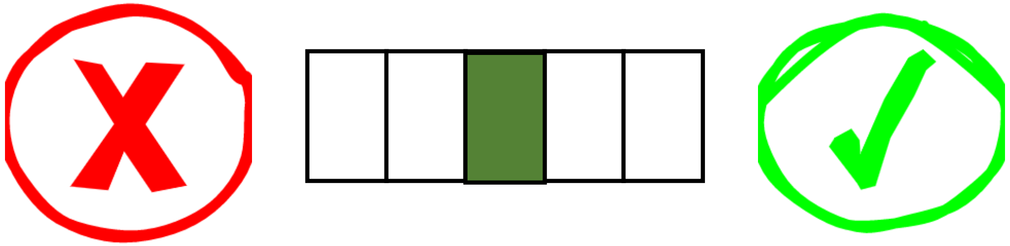
```
while(!game_over)
    action = pick_action(state)
    reward = act(action, state)
```

Single random agent, static environment

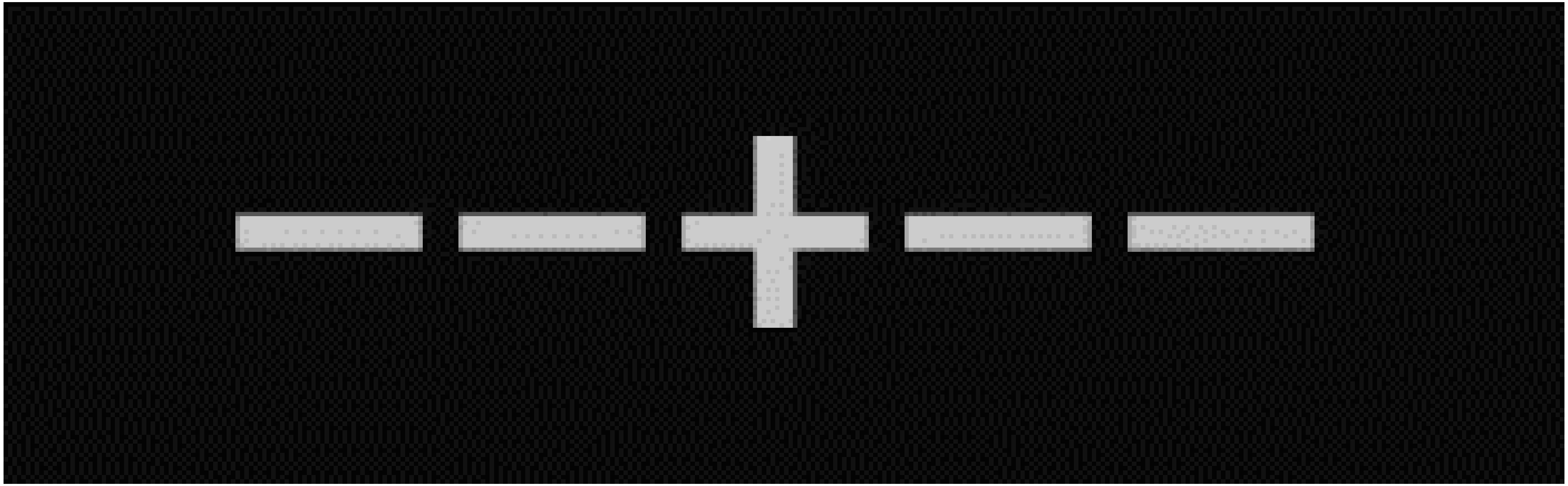
- An agent, shown by a green square, can **act**
  - By moving left or right, but not off the edge of the world
- Reward (score) is -1 for off the left, +1 for out off the right.
- No learning this time
  - Movement stochastic or random



```cpp
void rnd()
{
    constexpr size_t line_length = 5;
    int position = 2;
    std::mt19937 gen(std::random_device{}());
    std::uniform_int_distribution dist(-1, 1);

    draw(world(line_length, position));
    while (position > -1 && position < line_length)
    {
        auto action = dist(gen);
        while (action == 0) { //want -1 or +1 only
            action = dist(gen);
        }
        position += action;
    }
    draw(world(line_length, position)); //world is just a string to display for now
}
```



```
void draw(const std::string & s)
{
    using namespace std::chrono;
    std::cout << "\x1B[2J\x1B[H";
    std::cout << s;
    std::this_thread::sleep_for(1000ms);
}
```



A random walk

- Expect Left/Right : 50/50
- Example run: Lost 44, Won 56
- Average steps: 9.0
- Let’s **learn** now

Single agent with model, static environment

- Previously, move was -1 or +1, at random.
- Now we use
  - Pick action (left/right) based on **model**
    - Which can be a fixed action, like go right
  - Or do something random, with probability ε (epsilon)
- Hard code action first…
  - Then we'll let the agent find this out.
- Single agent, but has several goes (episodes)

General idea:

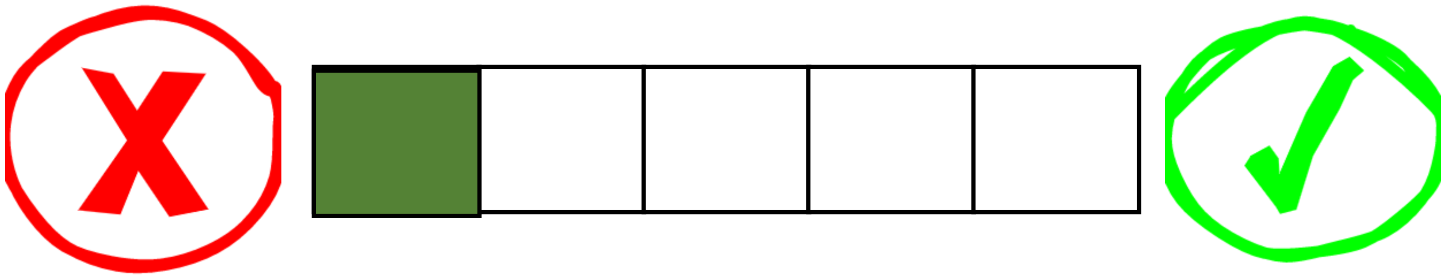
```
for_each(episode)
    env.reset()
    while(!game_over)
        action = pick_action(state)
        reward = env.step(action, state)
        // ignored,
        // but could be used to update model
```

```cpp
void fixed_model() {
    constexpr size_t line_length = 5;
    int position = 2;
    std::mt19937 gen(std::random_device{}());
    std::uniform_int_distribution<> dist(-1, 1);
    std::uniform_real_distribution<> prob(0.0, 1.0);
    constexpr double epsilon = 0.1;

    draw(world(line_length, position));
    while (position > -1 && position < line_length) {
        auto action = prob(gen) < epsilon ?
            dist(gen) : best_fixed_action(position);

        while (action == 0)
            action = dist(gen);
        position += action;
        draw(world(line_length, position));
    }
}
```

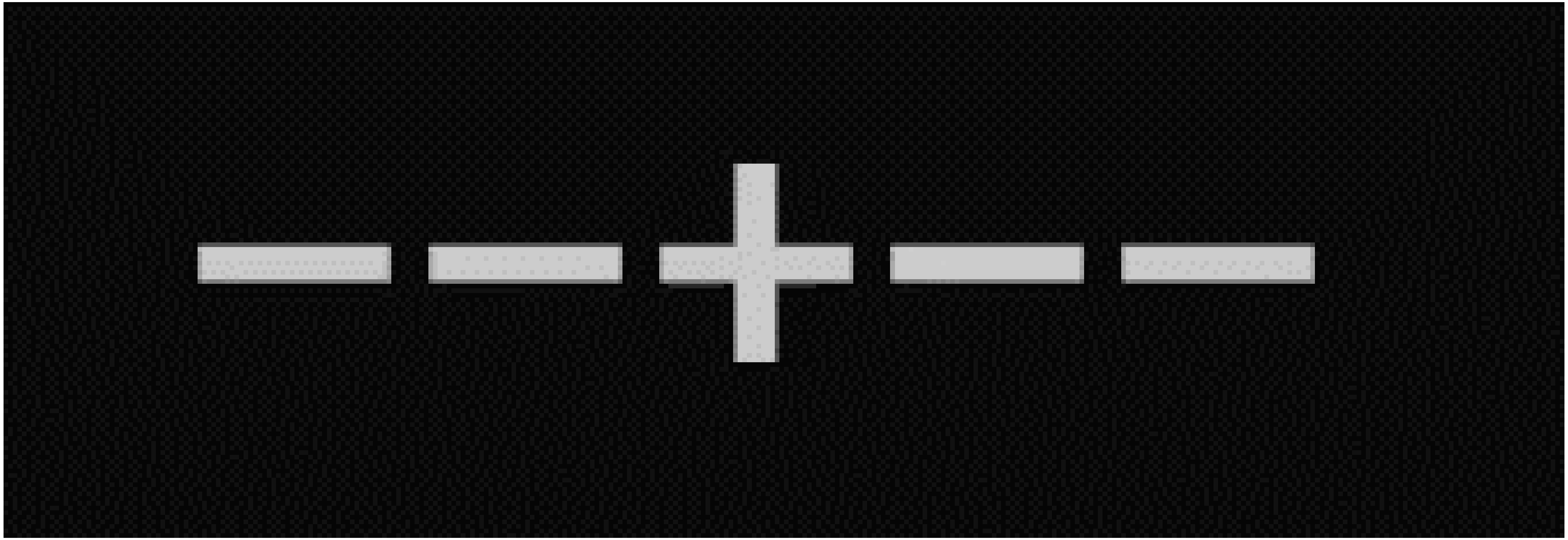
probability ε = 0.1 = $\frac{1}{10}$

$$action = \begin{cases} rnd, & p < \frac{1}{10} \\ best, & x \geq \frac{1}{10} \end{cases}$$

A model (or fixed action)



```
int best_fixed_action(int position)
{
    return 1; // Strong types would be better
}
```



Stats: (almost) everyone’s a winner

- Example run: Lost 0, Won 100
- Average steps: 3.28
- The agent only has a 50/50 chance to go left 1/10 times.
- It didn’t learn anything though
- We told it a model
- Can it figure this out on its own?

Now let's try to learn

- Previously we used
  - Pick action based on a model
  - Or did something random, with probability ε (epsilon)
- Now, we'll pick action for current state and store reward
  - Reward -1 for out on left, +1 for out on right, 0 otherwsie
  - lookup[(state, action)] -> reward
  - Key: position and action pair
  - Value: action
- Note: Picking an action based on previous rewards
  - Assumes the rewards are stationary
- `std::map<std::pair<int, int>, int> quality;`

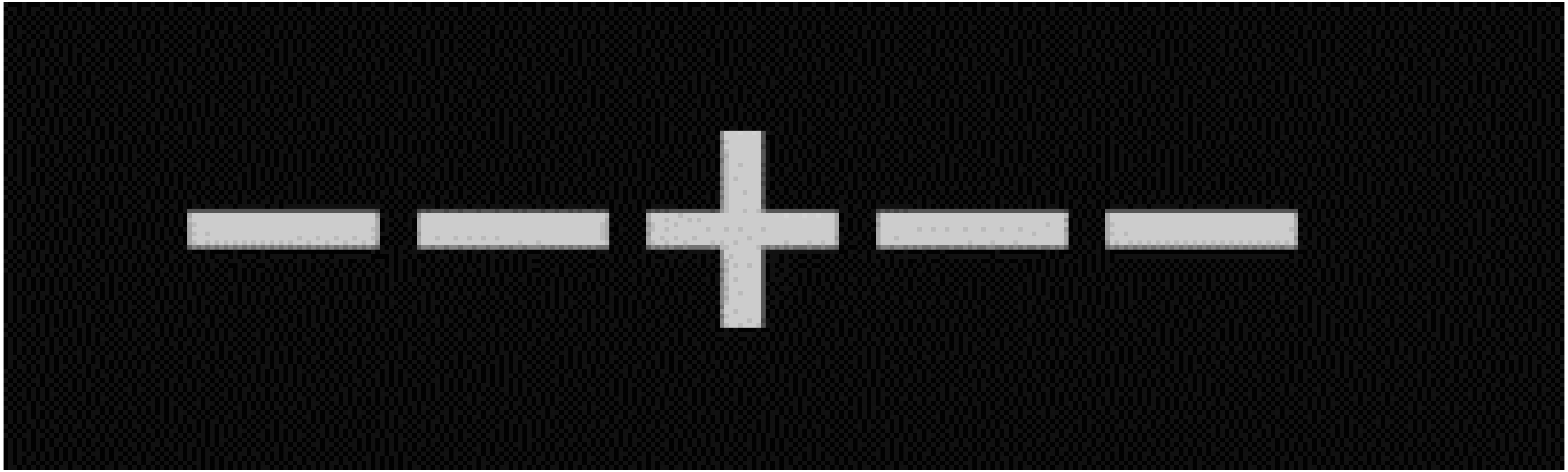
General idea:

```
Lookup quality{}
for_each(episode)
    env.reset()
    while(!game_over)
        action = pick_action(state, quality)
        reward = env.step(action, state)
        learn(action, state, reward)
```

```cpp
auto action = prob(gen) < epsilon ?
    dist(gen) : best_action(position, quality);

int best_action(int position,
        std::map<std::pair<int, int>, int> & quality)
{
    auto left = quality[{position, -1}]; //quality not const
    auto right = quality[{position, 1}]; //'cos a it's map

    if (left == right)
        return 0; // to say do a rnd, enum maybe clearer
    if (left > right)
        return -1; // left
    return 1; //right
}
```

```cpp
int reward(int position, size_t line_length)
{
    if (position < 0)
        return -1;
    else if (std::cmp_less(position, line_length))
        return 0;
    return +1;
}
```

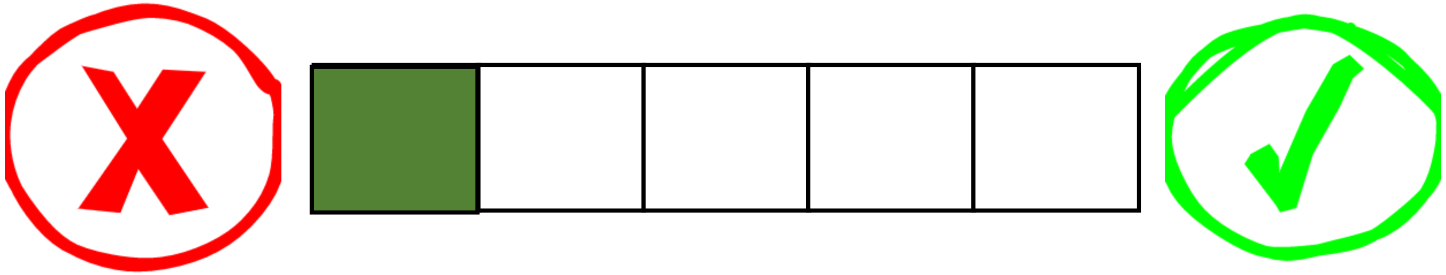
```cpp
void learn(int state, int action, int reward
    std::map<std::pair<int, int>, int>& quality)
{
    quality[{state, action}] += reward;
}
```



Stats

- Example run: Lost 11, Won 89
- Average steps: 11.88
- Recall, with the fixed model:
  - Example run: Lost 0, Won 100
  - Average steps: 3.28
- Better than pure random:
  - Example run: Lost 44, Won 56
  - Average steps: 9.0

Quality



```
best_action:
if (quality(left) == quality(right))
    return 0; // ? shrug
if (quality(left) > quality(right))
    return -1; // left
return 1; // right
```

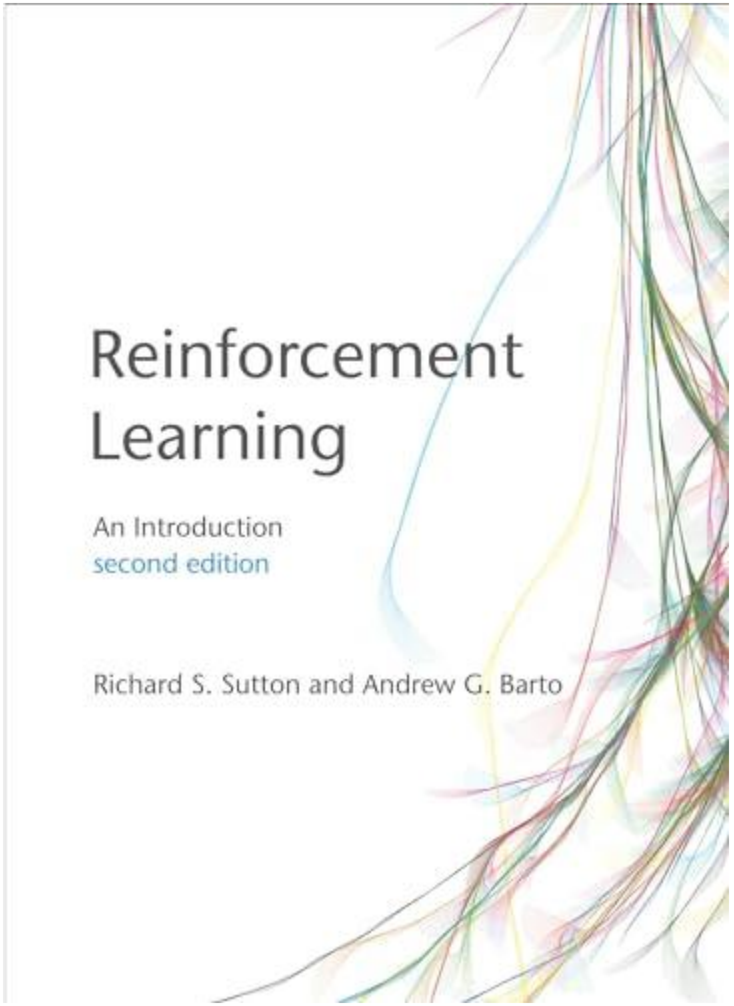
Position	Action	Quality
0	Left	-11
0	Right	0
1	Left	0
1	Right	0
2	Left	0
2	Right	0
3	Left	0
3	Right	0
4	Left	0
4	Right	0
5	Left	0
5	Right	89

So much nothing

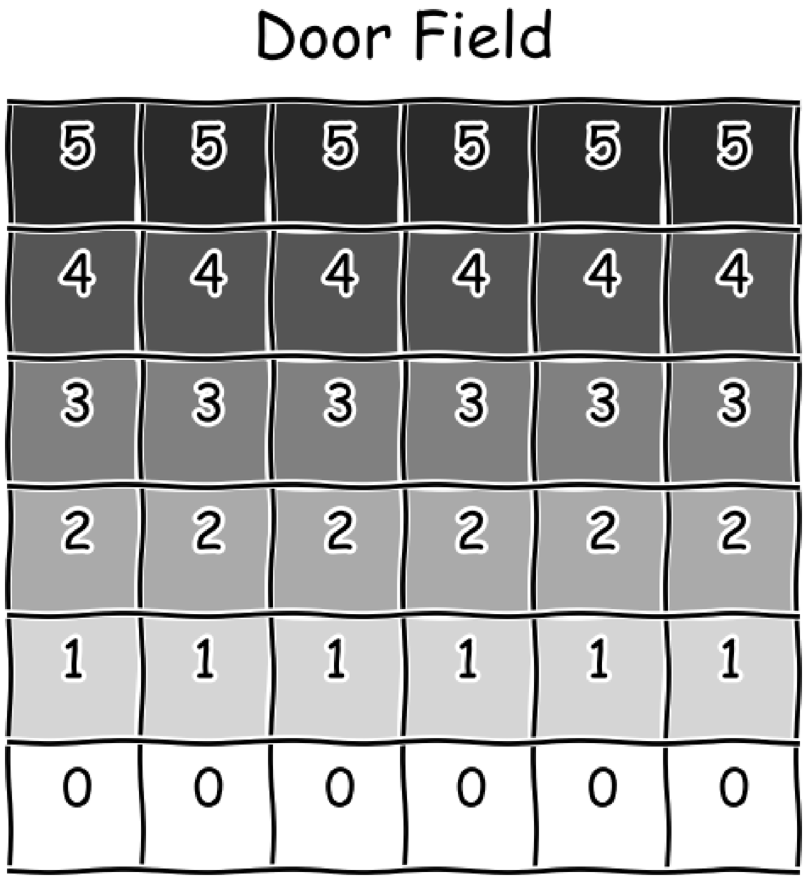
- Quality: -11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 89
- Look at all the 0s!
- It explores a bit
  - Epsilon means it sometimes fails
  - Greedy (best always) instead means it only steps off the left edge once
- Agent has learnt to avoid going left
  - BUT only knows one step right is better at the right hand edge

Quality Learning

- Q-learning
  - Chris Watkins, 1989 PhD
- One type of RL
- Doesn’t need a model
- The quality table uses last and next state
  - a % of the next known rewards add to the state
  - meaning newer rewards can be more important
  - so agents can learn in a dynamic env



Crowd your way out of a paper bag



https://www.youtube.com/watch?v=wlsbg5q0hO0&t=5s

But now the agent learns

- Recap: so much nothing
- Want to **learn**, without of floor field or model
  - The agent will build up this “field” through experience
  - This “field” is called a **policy** in RL: what action to take in a given state
- Want agent to discover to head in a direction
  - So not have lots of 0s in the middle
- One type of RL: Temporal difference (TD)
  - Notes reward in a given state and remembers what it can do in the new state
  - Including potential maximum reward
  - learn(state, action, reward) -> learn(last_state, action, reward, next_state)

Q-Learning

- Q-learning is "Off-policy TD Control"
  - Off-policy: estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed
  - SARSA is on-policy (estimates the return for state-action pairs assuming the current policy continues to be followed.)
- Uses the Bellman equation:
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
- Quality (for last state) becomes
  - previous quality + α (learning rate) times
    - Reward this time
    - + γ (discount/forgetting) times best possible from this state next state
    - Subtract previous quality
- Percolates next state back
  - So the zeros in the middle go

Q-Learning: It's all Greek to me

- **Epsilon** for random action versus best action based on Q-table.
  - Explore/exploit
- **Alpha** for learning
  - Weights the reward plus potential next reward
- **Gamma**
  - AKA "discount factor", beta (or lambda), or "impatience"
  - If 0, the agent is **myopic**, since it only uses the latest reward
- The Bellman (optimality) equation
  - Dates to 1950s, introduced in "Dynamic Programming" book
  - calls gamma the discount factor
  - https://en.wikipedia.org/wiki/Bellman_equation

General idea:

```
Lookup quality{}
for_each(episode)
    env.reset()
    while(!game_over)
        action = pick_action(state, quality)
        reward = env.step(action, state)
        q_learn(last_state,
                action, reward, next_state)
```

```cpp
void q_learn(int state, int action, int reward, int next_state,
    std::map<std::pair<int, int>, double>& quality)
{
    constexpr double alpha = 0.1;
    constexpr double gamma = 0.99;
    double predict = quality[{state, action}];
    auto v = quality
        | std::views::filter([next_state](auto kvp) {
            return std::get<0>(kvp.first) == next_state;
         });
    auto potential_best = v.empty() ? 0.0 :
        std::ranges::max_element(v, [](const auto& lhs, const auto& rhs) {
            return lhs.second < rhs.second;
        })->second;
    double target = reward + gamma * potential_best;
    quality[{state, action}] += alpha * (target - predict);
}
```

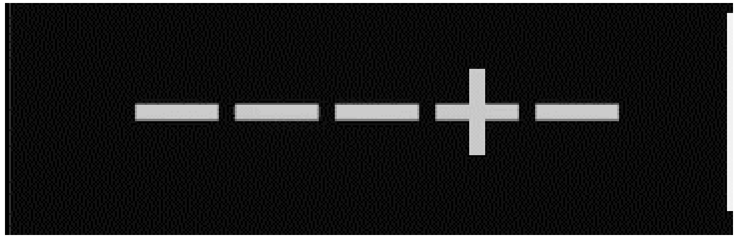
Stats:

- Mean steps 3.94
- Q-table
- (0, -1), -0.1
- (0, 1), 0.0265353
- (1, -1), 0
- (1, 1), 0.338015
- (2, -1), 0.0553337
- (2, 1), 0.97863
- (3, -1), 0.379363
- (3, 1), 0.989742
- (4, -1), 0.336458
- (4, 1), 0.99997

Recap

- Now mean steps is 3.94
- Average steps: 11.88 for lots of 0s in the middle
- Recall, with the fixed model:
  - Example run: Lost 0, Won 100
  - Average steps: 3.28
- Better than pure random:
  - Example run: Lost 44, Won 56
  - Average steps: 9.0

Demo



Time to go 2D

- Time to go 2D
- And tidy the code
  - Position was an `int`
- Lookup:
  - Was `std::map<std::pair<int, int>, double> quality;`
  - `std::map<std::pair<std::pair<int, int>, int>, double> (!)`
  - `Lookup<pair<Position, Action>, reward>`
- Reward function needs to change too

General idea (unchanged):

```
Lookup quality{}
for_each(episode)
    env.reset()
    while(!game_over)
        action = pick_action(state, quality)
        reward = env.step(action, state)
        q_learn(last_state,
                action, reward, next_state)
```

```cpp
struct Position {
    int x;
    int y;
    auto operator<=>(const Position&) const = default;
};

enum class Action {
    Shrug,
    Left,
    Right,
    Up,
    Down
};
using Lookup = std::map<std::pair<Position, Action>,
                        double>;
```

```cpp
double best_possible_reward_given_state(CharState state, LookupCharOnly& quality) {
    auto v = quality | std::views::filter([state](auto kvp) {
        return std::get<0>(kvp.first) == state; });
    return v.empty() ? 0.0 : std::ranges::max_element(v,
            [](const auto& lhs, const auto& rhs) { return lhs.second < rhs.second;}
        )->second;
}

void q_learn(CharState state, Action action, double reward,
    CharState next_state, LookupCharOnly& quality) {
    constexpr double alpha = 0.1;
    constexpr double gamma = 0.99;
    const double predict = quality[{state, action}];
    const double potential_best = best_possible_reward_given_state(next_state, quality);
    double target = reward + gamma * potential_best;
    quality[{state, action}] += alpha * (target - predict);
}
```

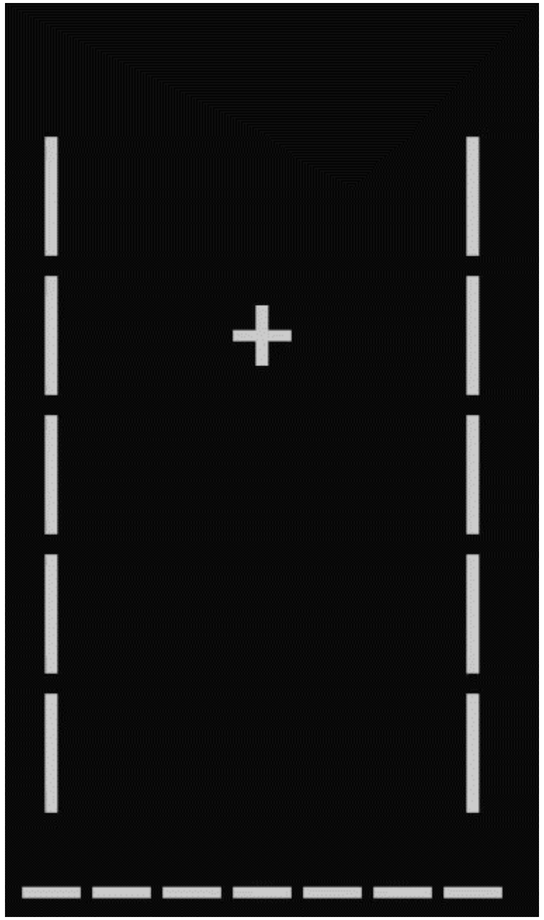
```cpp
Action pick_best_action(CharState state, LookupCharOnly& quality) {
  static std::mt19937 gen(std::random_device{}());
  std::vector<double> weights;
  const std::vector<Action> actions{Action::Up, Action::Down, Action::Left, Action::Right};

  for (auto act : actions)
    weights.push_back(quality[{ state, act }]);

  auto [smallest_pos, biggest_pos] = std::minmax_element(weights.cbegin(), weights.cend());
  auto smallest = *smallest_pos;
  auto biggest = *biggest_pos;

  std::transform(weights.begin(), weights.end(), weights.begin(),
        [biggest](const double& x) { return x == biggest ? 1.0 : 0.0; }
  );
  std::discrete_distribution<int> dd{ weights.cbegin(), weights.cend() };
  return actions[dd(gen)];
}
```

Or easier to see:

- Pick best action
  - Or random choice if two or more equally good
- Don’t forget, this only happens sometimes, based on epsilon:

```
auto action = prob(gen) < epsilon ?
    static_cast<Action>(dist(gen)) :
    pick_best_action(position, quality);
```

Out of a paper bag!



Stats

Won 20, lost 30

Q-table

(0, 0: Left), -0.1

(0, 0: Down), -0.1

(0, 1: Left), -0.1

(0, 2: Left), -0.19

(0, 3: Left), -0.271

(0, 4: Up), 0.499001

(1, 0: Down), -0.3439

(1, 4: Right), 0.0296703

(1, 4: Up), 0.3994

(2, 0: Down), -0.271

(2, 4: Up), 0.697903

(3, 0: Down), -0.19

(4, 0: Right), -0.3439

(4, 0: Down), -0.19

(4, 1: Right), -0.271

(4, 2: Right), -0.1

(4, 3: Right), -0.19

(4, 4: Right), -0.0703297

(4, 4: Up), 0.3994

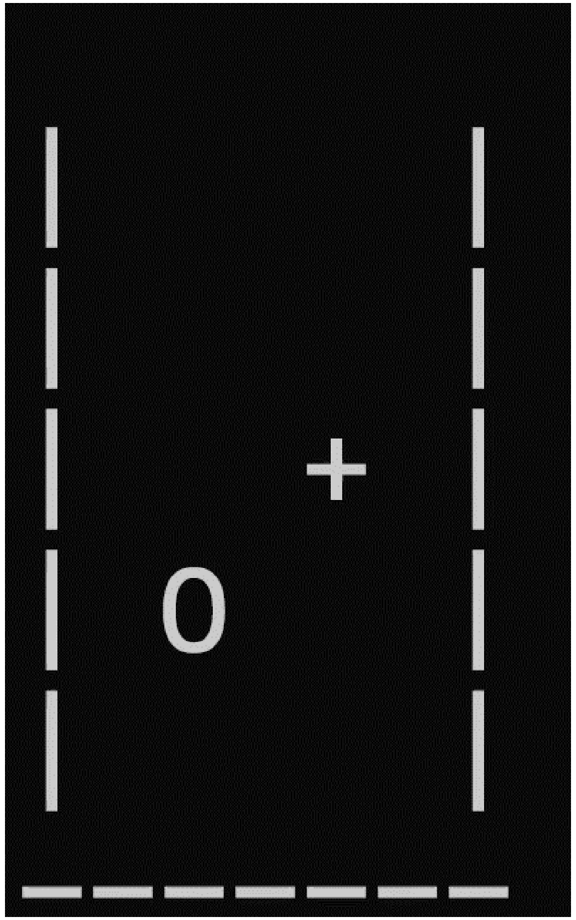
What about an obstacle?

- We want to do snake
  - eventually
- Let’s start with something to avoid, like ‘O’ somewhere
  - Then turn it into an apple to eat
  - So the single + becomes a snake
  - (But we’ll have to deal with dynamic env then)

```
|       |
|       |
|    +  |
| O     |
|       |
--------
```

Some obstacles

```
std::deque<Position> obstacles{}; // Demo {1,1}
bool game_over() const {
    auto it = std::ranges::find(obstacles, pos);
    return it != obstacles.end()
        || pos.x < 0 || pos.x >= max_x || pos.y < 0 || pos.y >= max_y;
}
int step(Action action) {
    pos = perform_action(pos, action);
    if (game_over())
    {
        return pos.y >= max_y ? 1 : -1; // Allowed out of bag - reward in that case
    }
    return 0;
}
```


+
0

One obstacle, greedy action

Won 16, lost 34

(0, 0: Left), -0.1

(0, 0: Down), -0.1

(0, 1: Right), -0.1

(0, 2: Left), -0.271

(0, 3: Left), -0.1

(0, 4: Left), -0.1

(1, 0: Up), -0.1

(1, 0: Down), -0.271

(1, 2: Down), -0.468559

(1, 4: Up), 0.3994

(2, 0: Down), -0.271

(2, 1: Left), -0.468559

(2, 4: Up), 0.697903

(3, 0: Down), -0.1

(4, 0: Right), -0.1

(4, 0: Down), -0.19

(4, 1: Right), -0.1

(4, 2: Right), -0.1

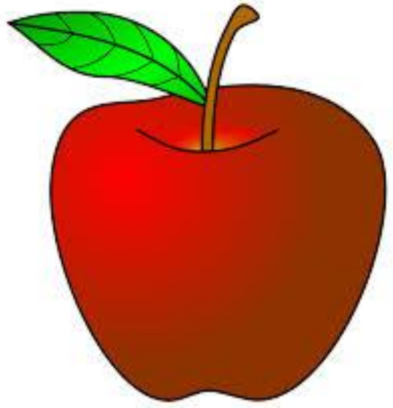
(4, 3: Right), -0.1

(4, 4: Up), 0.499001

Left: -0.1	Up: 0.3994	Up: 0.697903		Up: 0.499001
Left: -0.1				Right: -0.1
Left: -0.271	Down: -0.268559			Right: -0.1
Right: -0.1		Left: -0.468559		Right: -0.1
Left: -0.1, Down -0.1	Up: -0.1, Down -0.271	Down: -0.271	Down: -0.1	Right: -0.1, Down: -0.19

Dynamic env

- What if the environment is dynamic?
  - Change the obstacle to an apple
  - New an apple appears
    - Agent needs to forget old position
- Snake will grow when it eats the apple
  - But we’ll just respawn the apple first
    - And grow the snake later
  - New reward function
  - And new danger
- We’ll try the same lookup for the Q-table
  - Then a new lookup state



General idea (unchanged, apart from env):

```
Lookup quality{}
for_each(episode)
    env.reset()
    while(!game_over)
        action = pick_action(state, quality)
        reward = env.step(action, state)
        q_learn(last_state,
                action, reward, next_state)
```

Env changes: no proper snake (yet)

```
int step(Action action) {
    pos = perform_action(pos, action);
    if (game_over()) {
        return pos.y >= max_y ? 1 : -1;
    }
    if (apple == pos) {
        while (apple == pos) {
            apple = random_apple(max_x, max_y);
        }
        return 1;
    }
    return 0;
}
```

Demo (if time)

- Debug\TwoDime.exe a

(0, 0: Left), -0.559394
(0, 0: Right), 0.0523588
(0, 0: Up), 0.0767104
(0, 0: Down), -0.174421
(0, 1: Left), -0.480107
(0, 1: Right), 0.0223568
(0, 1: Up), 0.0434685
(0, 1: Down), 0.0764805
(0, 2: Down), 0.0979209
(0, 3: Left), -0.1
(1, 0: Down), -0.569533
(1, 1: Right), 0.00959766
(1, 1: Up), 0.00970387
(1, 1: Down), 0.0968491
(1, 4: Right), 0.0296703
(1, 4: Up), 0.2997

(2, 0: Left), 0.0980189
(2, 0: Down), -0.252313
(2, 2: Left), 0.392291
(2, 2: Down), 0.0392666
(2, 3: Left), 0.00988021
(2, 3: Up), 0.198903
(2, 4: Up), 0.697903
(3, 0: Down), -0.19
(3, 1: Left), 0.0999
(3, 1: Down), 0.0099
(3, 2: Left), 0.0099
(3, 2: Right), 0.0989055
(3, 2: Up), 0.0186749
(3, 4: Up), 0.1

(4, 0: Right), -0.19
(4, 2: Right), -0.1
(4, 3: Right), -0.271
(4, 4: Right), -0.1

Massive state space

- 5 by 5
  - plus bag edges
  - possible O or + anywhere
  - will slowly forget last apples place
- In fact, I can’t tell where the apples were from the Q-table!
- Does the position matter?
- Character up, down, left, right matters
  - And maybe direction to food
  - Penalise if the agent moves further away

A new lookup

```
struct CharState
{
    char cu;
    char cd;
    char cl;
    char cr;
    int horizontal; // to food -1, 0, +1
    int vertical; // to food -1, 0, +1
    auto operator<=>(const CharState&) const = default;
};
using LookupCharOnly =
    std::map<std::tuple<CharState, Action>, double>;
```

And a snake game (finally!)

```cpp
using Snake = std::deque<Position>;
class Game;
private:
int max_x{};int max_y{};
Snake snake_{ {max_x / 2, max_y / 2} };
Position apple_{respawn_apple()};
std::function<Position()> spawn_apple;
void respawn_apple() {
    do {
        apple = spawn_apple();
    }
    while (std::ranges::find(snake_, apple)
        != snake_.end());
}
bool game_over_{ false };
```

```cpp
public:
    Game(int max_x, int max_y,
        std::function<Position()> fn);

    [[nodiscard]] bool move(Direction dir);
    bool game_over() const { return game_over_; }
    void reset(Snake snake);
```

Move snake:

```cpp
using Snake = std::deque<Position>;
bool SnakeLib::Game::move(Direction dir) {
    if (!SnakeLib::move(snake_, dir, max_x, max_y)) {
        game_over_ = true;
    }
    else if (eat(snake_.front(), apple)) {
        apple = respawn_apple();
        return true;
    }
    else {
        snake_.pop_back();
    }
    return false;
}
```

And a new environment with a game…

```cpp
class Environment
{
    SnakeLib::Game game;
    SnakeLib::Snake start_snake{game.Snake()};
public:
    Environment(const SnakeLib::Game& game) : game(game) {};
    bool game_over() const { return game.game_over(); }
    void reset() { game.reset(start_snake); };
    FoodDirection food_direction() const;
    int step(Action action);
    CharState state() const;
    SnakeLib::Game& Game() { return game; }  // leaky but hey
};
```

… rewards

`int Environment::step(Action action)`

- `−2 for game over (including out of bag)`
- `+2 for ate apple`
- `−1 for further from food`
- `+1 for nearer food`

(Some) Stats

(, , , , [0, -1]: Left), -0.161781

(, , , , [0, -1]: Right), -0.1

(, , , , [0, -1]: Up), 1.27213

(, , , , [0, -1]: Down), 7.40003

(, , , , [0, 1]: Left), 2.56456

(, , , , [0, 1]: Right), 0.466925

(, , , , [0, 1]: Up), 7.22619

(, , , , [1, -1]: Left), 2.1945

(, , , , [1, -1]: Right), 3.04448

(, , , , [1, -1]: Up), 2.35066

(, , , , [1, -1]: Down), 7.68758

(, , , , [1, 0]: Left), -0.1

(, , , , [1, 0]: Right), 7.68254 //…

(, , ,O, [1, 0]: Left), 2.18344
(, , ,O, [1, 0]: Right), 7.42734
(, , ,O, [1, 0]: Up), 1.6151
(, , ,O, [1, 0]: Down), 0.0784815
(, , ,|, [-1, -1]: Left), 0.665528
(, , ,|, [-1, 0]: Up), -0.0901
(, , ,|, [-1, 1]: Left), 2.59265
(, , ,|, [-1, 1]: Down), -0.1
(, , ,|, [0, 1]: Left), 0.37943
(, , ,|, [0, 1]: Right), -0.153379
(, , ,|, [0, 1]: Up), 6.61384
(, , ,|, [0, 1]: Down), 0.081203

(, ,+, , [-1, -1]: Left), 0.276949
(, ,+, , [-1, -1]: Right), -0.1
(, ,+, , [-1, -1]: Up), 0.0830456
(, ,+, , [-1, -1]: Down), 7.50895
(, ,+, , [-1, 0]: Left), 3.72608
(, ,+, , [-1, 0]: Right), 0.052774
(, ,+, , [-1, 0]: Down), -0.126848
(, ,+, , [-1, 1]: Left), -0.0901
(, ,+, , [-1, 1]: Right), 0.27479
(, ,+, , [-1, 1]: Up), 8.14921
(, ,+, , [-1, 1]: Down), -0.228743

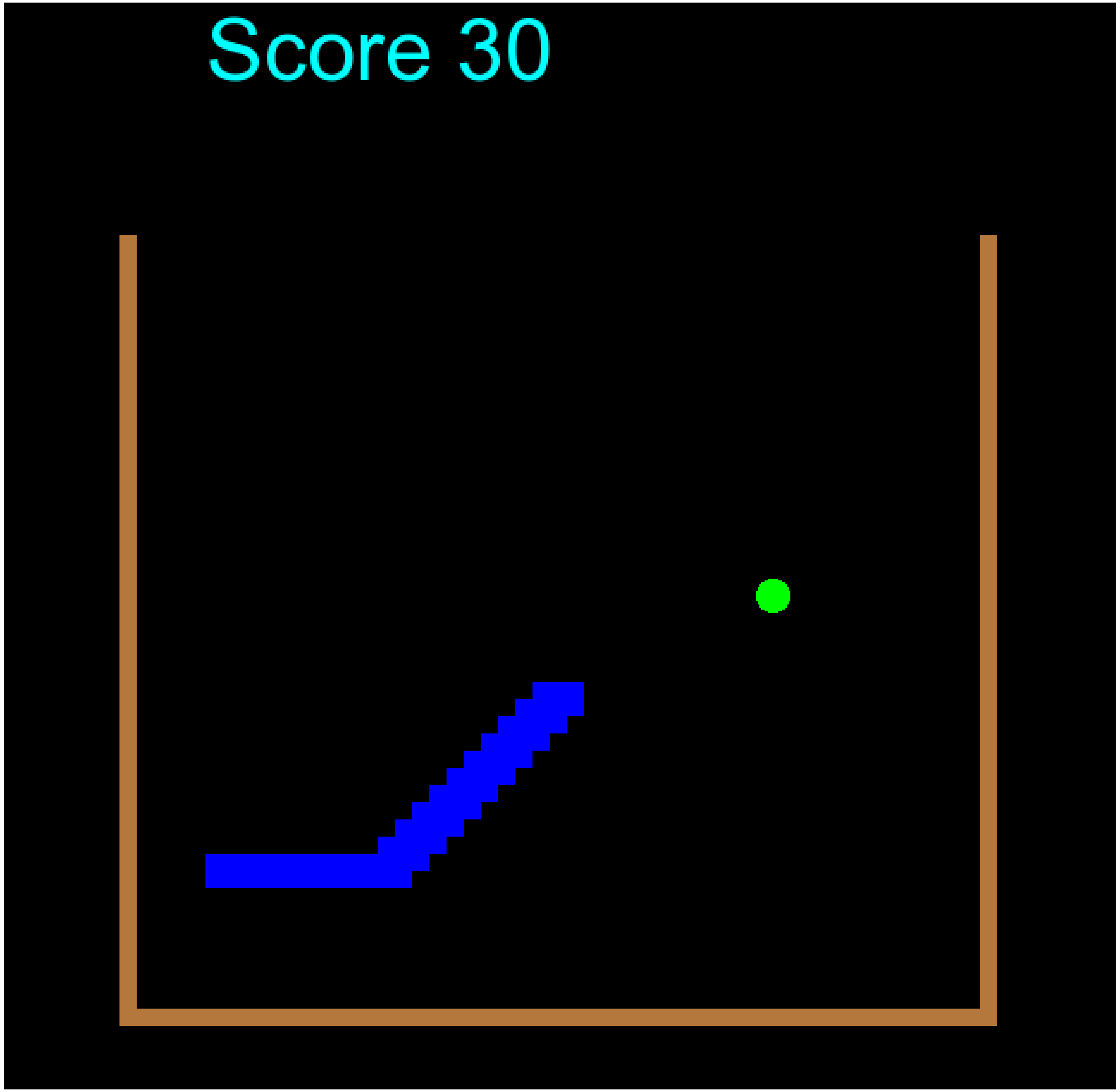
Demo

- Maybe time for a demo? Use Apples.exe q_table_1000.txt

```
|   +*+|
|O   ++|
|    ++|
|    ++|
|    ++|
-------
Won 0, lost 1
Scores
20
```

SNAKE!!!!!!!!!!

- Configure gird size
  - Started 5 by 5,
  - Then try 50 by 45
- Save the q_table
  - So we can read in a trained version
  - And use this to play Snake


Score 30

Demo

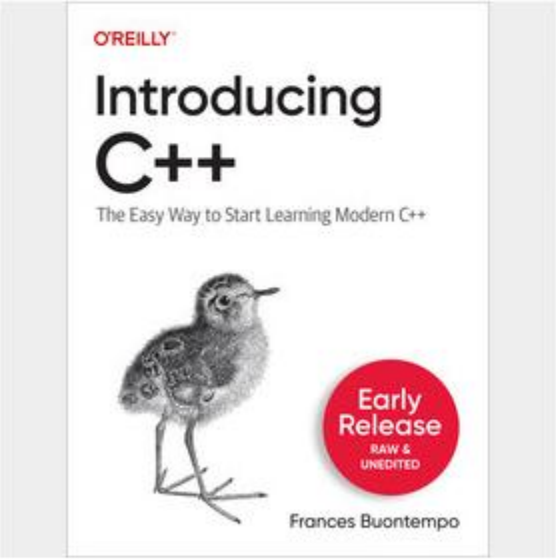
- Debug\DemoTrainedRL.exe ..\q_table_large_10000.txt
- Also a short screen capture here: https://www.linkedin.com/posts/francesbuontempo_i-have-attempted-to-teach-my-machine-how-activity-7303789398104395776-CX8n/

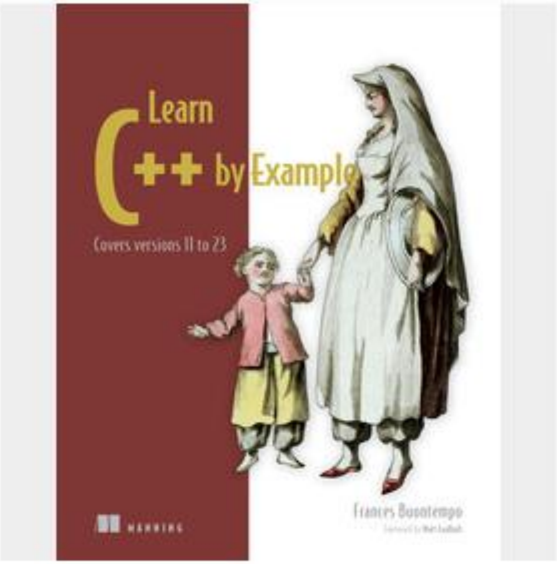
Summary

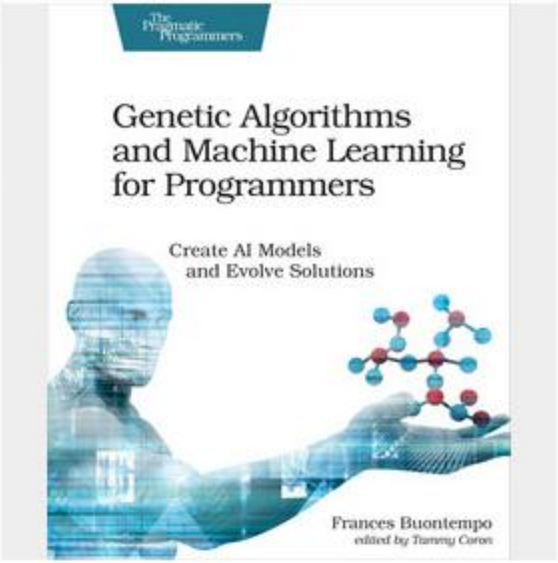
- Reinforcement learning is one type of "AI"
- It doesn't need labelled data
  - Simple versions like Q-learning store data in tables
  - A kind of Markov decision process, mapping states to actions
    - https://accuconference.org/2025/session/a-very-small-language-model (Jez's talk)
- I used a temporal difference Q-learning approach
  - There are other approaches
- It can discover how to play games
  - Can be used in the "fine tuning" of LLMs
  - And do more useful stuff
    - Compiler optimization, logistics, power grid management, video compression

Other things to go play with

- https://play.battlesnake.com/
- https://compilergym.com/getting_started.html
  - a toolkit for applying reinforcement learning to compiler optimization tasks.
- OpenQI's Gym: https://gymnasium.farama.org/index.html
- Arcade Learning Environment (ALE): https://ale.farama.org/environments/
  - Asteroids, PacMan…
  - Env can be RGB image
- Deep RL
  - Started the Atari games learning
  - Uses a convolutional neural network instead of a Q-table.
  - Playing Atari with Deep Reinforcement Learning, 2013, https://arxiv.org/abs/1312.5602
- Deep Reinforcement Learning in Pac-man
  - https://www.youtube.com/watch?v=QilHGSYbjDQ







Introducing C++

Learn C++ by Example

Genetic Algorithms and Machine Learning for Programmers

https://www.oreilly.com/library/view/introducing-c/9781098178130/

https://mng.bz/1aVV

https://pragprog.com/titles/fbmach/genetic-algorithms-and-machine-learning-for-programmers/