Machine Learning – Testing and Artificial Intelligence

15-110 – Wednesday 11/22

Announcements

- **Office hours today are in GHC 4109**
- Check6-1 grades are released
  - **Make sure to view your feedback on the programming part!**
- Check6-1 Revisions due **Monday 11/25 at noon**
  - This is the final item with a revision deadline
- Check6-2 due **Monday 11/25 at noon**
  - Don't forget to uncomment test cases at bottom of file!
  - Check6-2 / Hw6 autograded functions will **not** be manually graded for partial credit
- Monday is the last quizlet of the semester!

Learning Goals

- Describe how **training**, **validation**, and **testing** are used to build a model and measure its performance

- Recognize how AIs attempt to achieve **goals** by using a **perception, reason,** and **action** cycle

From Training to Testing

Once we've trained a model using machine learning, we may want to evaluate that model to see how well it actually works.

We can do this by **testing** the model using the test data held in reserve.

Testing Machine Learning Models

Training Data, Validation Data, Testing Data

Once we've trained a model, we can use that model to make predictions about future data that it has not seen.

We already separated our data into two sets: **training** data vs. **testing** data.

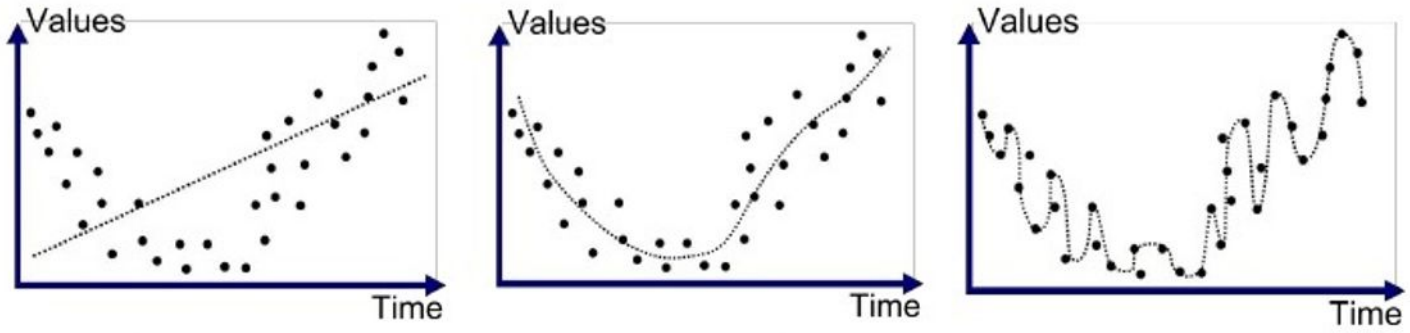
Additionally, **validation** data can help make the training work as well as possible before the final test is done.

Too Much Data Can Cause Overfitting

The **training data** is normally composed of the majority (maybe 70%) of the available dataset. More training data typically yields greater **accuracy.**

This can go wrong: **overfitting** occurs when the model identifies patterns that only exists within the training data, not in the general population.

Overfitting can result in a model performing very well on training data, but poorly on test data.



Underfitted Good Fit/Robust Overfitted

Validation Data Identifies Overfitting

We can reduce overfitting by using **validation data**. This is a subset of the data (maybe 15%) that is **not** used when training the model. Instead, it will be used to **validate** the model during training.

Training should continue while the accuracy on the validation data continues to increase.

Training should *stop* when the model's accuracy on the validation data begins to decrease.

Testing Data Provides Final Results

When the programmer thinks they've achieved an optimal model, the **testing data** is used to determine how accurate that model actually is. This is a portion of the data (maybe 15%) that was set aside at the beginning and never used during the training or validation process.

The model is run on the test data only **once,** after training. The accuracy on the training data is the accuracy of the model.

You cannot train on your testing data if you want a fair test of the model!

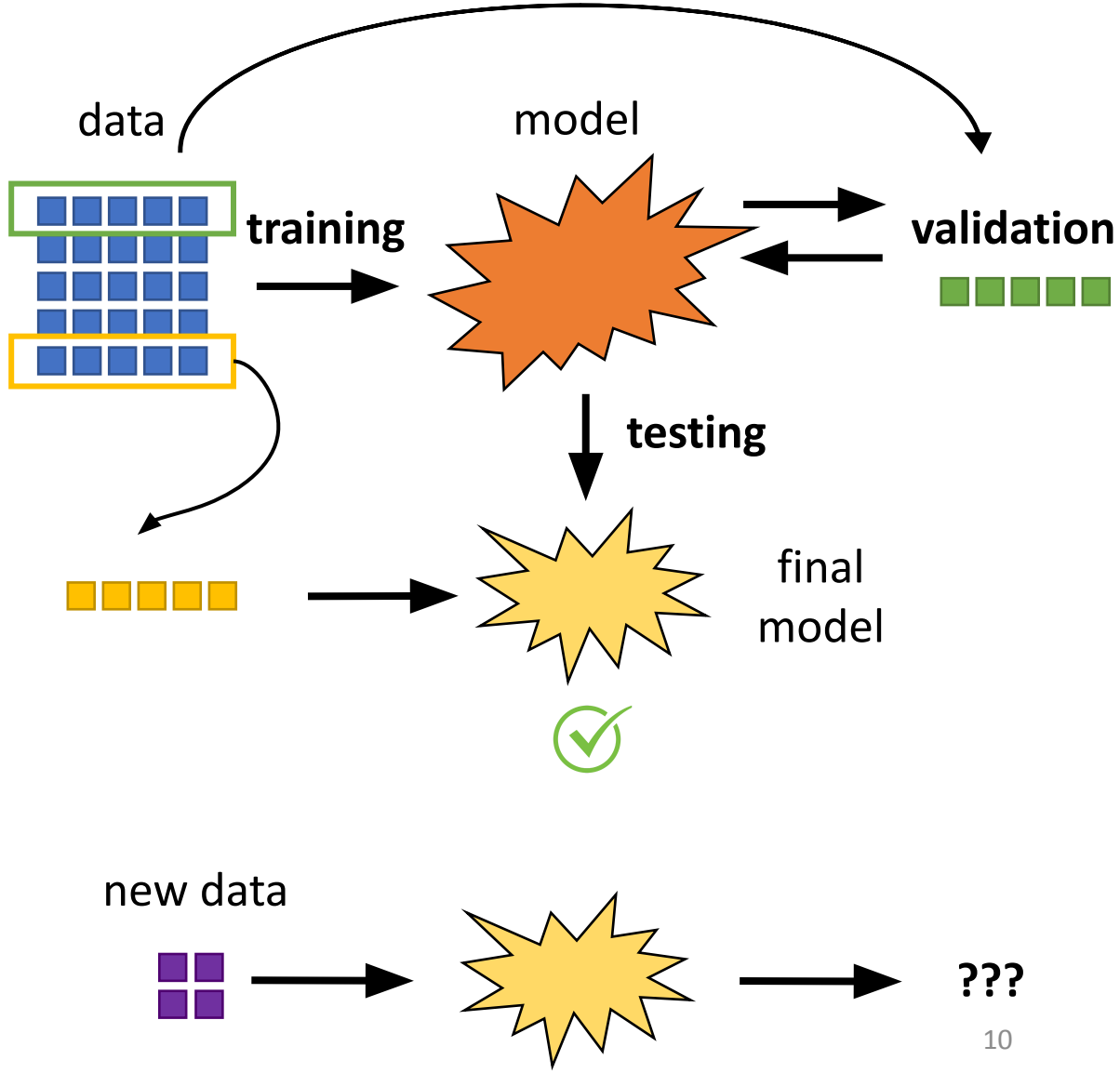
Example: Bad Training Process

What happens if we use our validation and test data to train as well?

The algorithm will get the opportunity to observe patterns in the additional data. It will optimize the model to include those patterns, even through validation.

When the model is tested, it will appear accurate because the model was optimized for this data.

But if we try to use the model on new, unlabeled data, the patterns may no longer be valid.



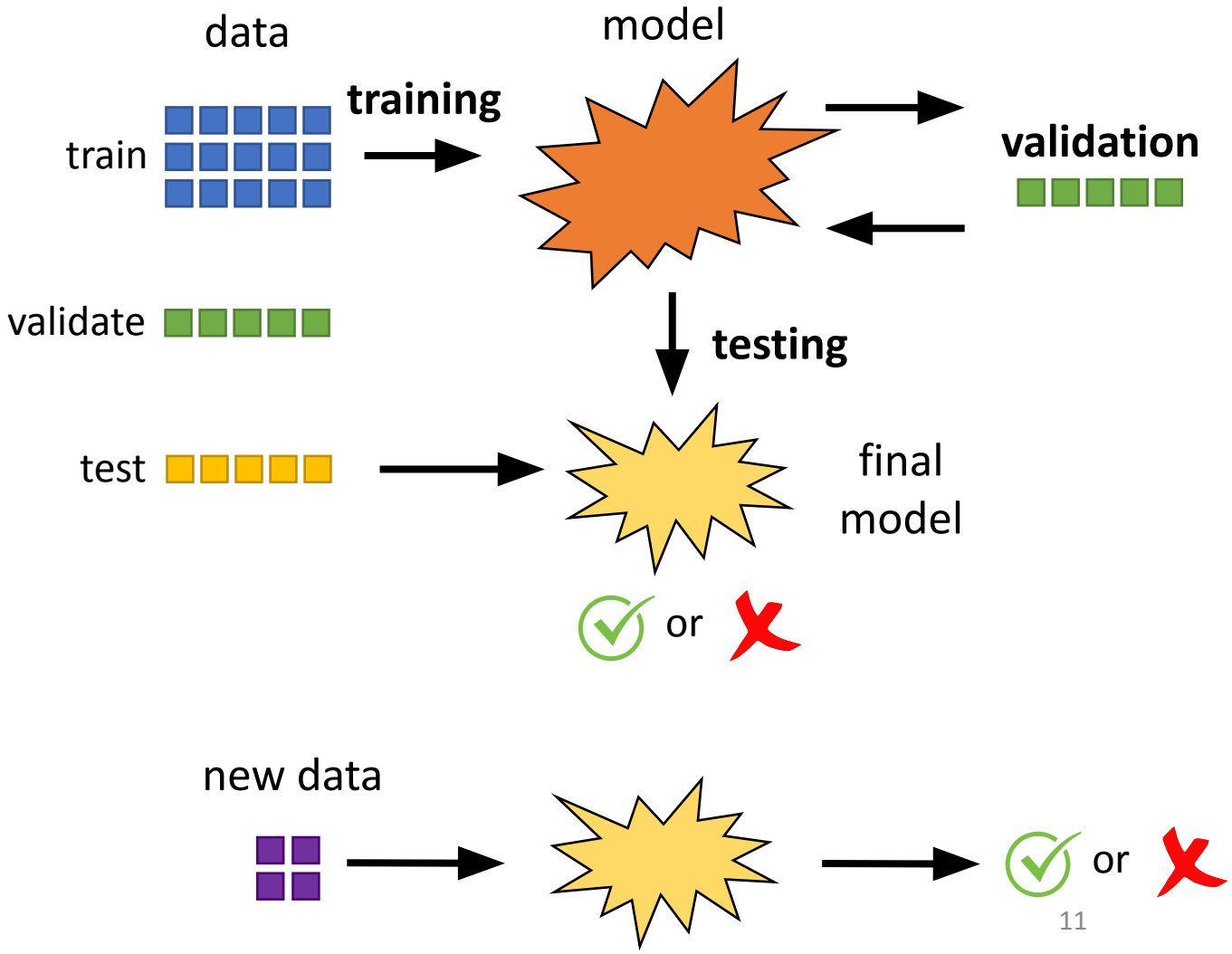
Example: Good Training Process

A better process: split the data into training, validation, and testing sets.

We'll train on **only** the training set and repeatedly test on the validation set. We stop training when the validation accuracy decreases, to prevent overfitting.

When we're done, we'll test on the test set **once**. That produces our final result.

Testing on new data should have about the same accuracy as the test data, since the model never saw the test data before.



Machine Learning Supports Artificial Intelligence

Now that we have machine learning algorithms, what can we do with them?

One option is to use them to support **artificial intelligence**. Let's talk more about what that means!

Artificial Intelligence

What is Artificial Intelligence?

Artificial Intelligence (AI) is a branch of computer science that studies techniques which allow computers to do things that, when humans do them, are considered evidence of intelligence.

However, it's extremely hard to build a machine with **general intelligence** (that is, a machine that can do everything a human can do). We believe we are still far away from this goal.

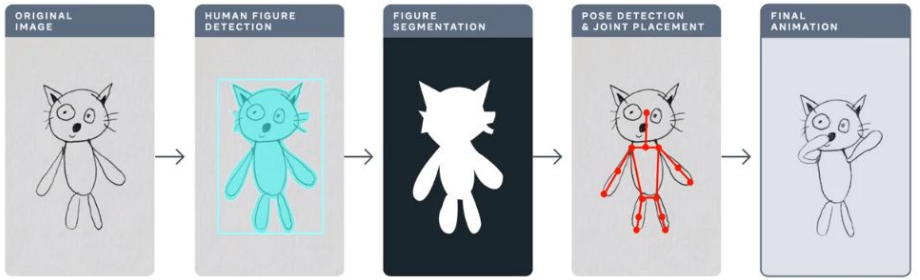
Most modern AI applications are **specialized**; they do one specific task, and they do it very well. We call an AI application trained for a specific task an agent.

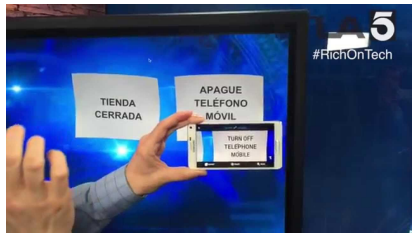
Examples of AI Agents

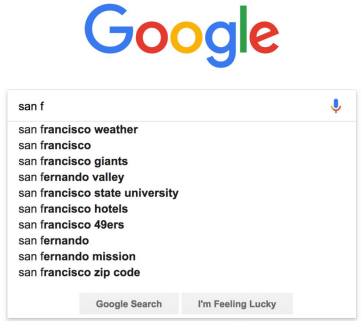
We've built AI agents that can play games, run robots, and animate children's drawings.

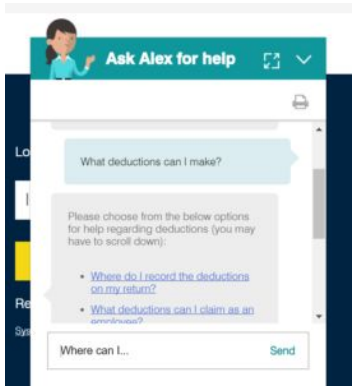
AI is also used to translate text, predict what you'll type, and answer questions on websites.

What do these agents have in common? Each agent we build has a specific **goal**, the thing it is trying to do.









What about Large Language Models (LLMs)?

Large Language Models (LLMs) like ChatGPT and Gemini are *more* general than their predecessors and can emulate a wider variety of human-like tasks, *but*:

- Training requires massive data and massive computational power
- Running the model also requires massive computational power
- Mistakes are still common, but harder to identify and reason about

Perception, Reason, and Action

Perception, Reason, and Action

Most AI agents attempt to reach goals by cycling through three steps: **perceive** information, **reason** about it, then **act** on it.

This is similar to how humans and animals work! We constantly take in information from our senses, process it, and decide what to do (consciously or unconsciously).

Perception: What Data Can Be Gathered?

First, the agent needs to **perceive** information about the state of the problem its solving.

This can include data given directly by the user and contextual information about past actions the user has taken.

An autocomplete agent might observe what the user is currently typing *and* what they've typed before.

Some agents can also perceive information through **sensors**, like cameras, microphones, accelerometers, and more



Reason: Create a plan for the future

Second, the AI agent needs to **reason** about the data it has collected, to decide what should be done next to move closer to the goal.

Reasoning uses **algorithms**, including **machine learning algorithms**. The agent often creates a **model representation** of the world based on the task it needs to solve and the data it has collected so far. It can then search through all the possible actions it can take to inform its decision.

A general goal of reasoning is to make correct decisions **quickly**. (For example, this is especially important for self-driving cars.)

Action: Follow through with the plan

Finally, the AI agent needs to **act**, to produce a change in the state of the problem.

Actions don't need to reach the goal *immediately*, and often can't. It is often sufficient to move closer to the goal, and then repeat the perception, reason, action cycle.

Agents that interface with the real world (robots) use **actuators** to make changes. This often involves the sub-fields of **kinematics, dynamics, and control**.

Example: IBM Watson

IBM's AI agent Watson was designed to play (and win!) the game Jeopardy. Its **goal** was to answer Jeopardy problems with a question.

Watson **perceived** the questions by receiving them as text, then broke them down into keywords using natural language processing.

It used that information to search for relevant information in its database. Watson used **reasoning** to determine how confident it was that the answer it found was correct.

If Watson decided to answer, it would **act** by organizing the information into a sentence, then pressing the buzzer with a robotic 'finger'.

Search Supports Artificial Intelligence

In Watson (and many other artificial intelligence applications), the key to being able to perceive and act quickly lies in **fast search algorithms**.

Being able to search quickly makes it possible for an AI agent to look through hundreds of thousands of possible actions to find which action will work best.

We've discussed many data structures and algorithms to support search already. The slides at the end of this deck describe two concepts used by AI agents to support fast search- **game trees** and **minimax**.

Learning Goals

- Describe how **training**, **validation**, and **testing** are used to build a model and measure its performance

- Recognize how AIs attempt to achieve **goals** by using a **perception, reason,** and **action** cycle

Extra Topics:
Game Trees and Minimax

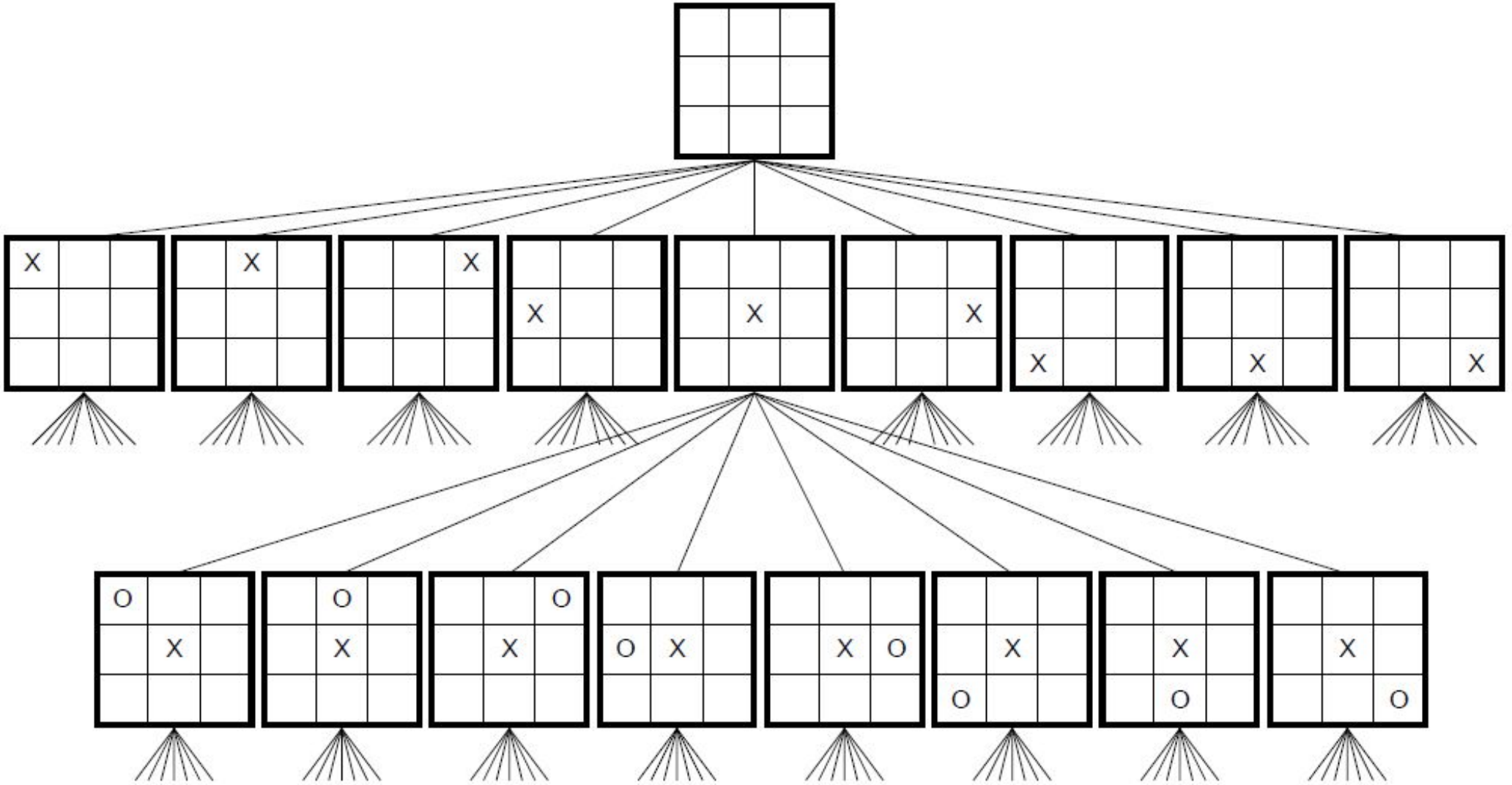
(You are not required to know this for the final exam)

Game Trees Represent Possible World States

To search data about possible actions and results quickly, an AI agent first needs to organize that data in a sensible way. Let's focus on a simple example: a two-player game between an AI agent and a human.

A game tree is a tree where the nodes are **game states** and the edges are **actions** made by the agent or the opposing player. Game trees let the agent represent all the possible outcomes of a game.

For example, the game tree for Tic-Tac-Toe looks like this...



Full board here: https://xkcd.com/832/

Reading a Game Tree

The **root** of a game tree is the current state of the game. That can be the start state (as in the previous example), or it can be a game state after some moves have been made.

The **leaves** of the tree are the final states of the game, when the AI agent wins, loses, or ties.

The **edges** between the root and the first set of children are the possible moves the agent can make. Then the next set of edges (from the first level of children to the second) are the moves the opponent can make. These alternate all the way down the tree.

Game Trees are Big

How many possible outcomes are there in a game of Tic-Tac-Toe?

Let's assume that all nine positions are filled. That means the **depth** of the tree is 10 (there are nine moves, so we count the root + 9 results of actions). There are 9 options for the first move, 8 for the second (for each of those nine states), 7 for the third, etc... that's **9!**, which is 362,880.

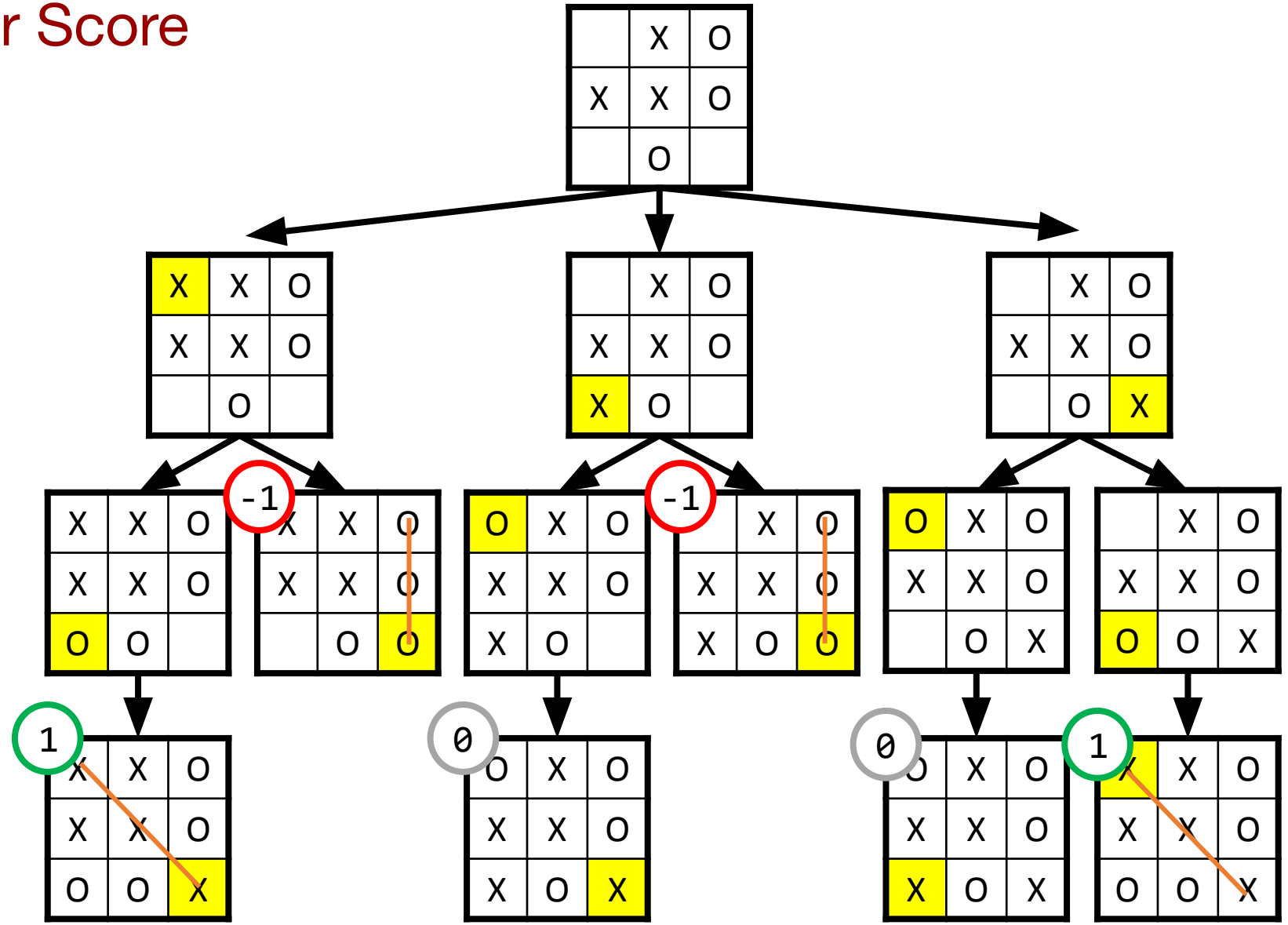
This number is a bit larger than the real set of possibilities (some games end early), but it's a good approximation.

How can the agent choose the best set of moves to make out of all these options?

Minimax Optimizes for Score

The **minimax** algorithm can be used to maximize the final 'score' of a game for an AI agent.

In Tic-Tac-Toe, we'll say that the score is 1 if the computer wins, 0 if there's a tie, and -1 if the human wins.



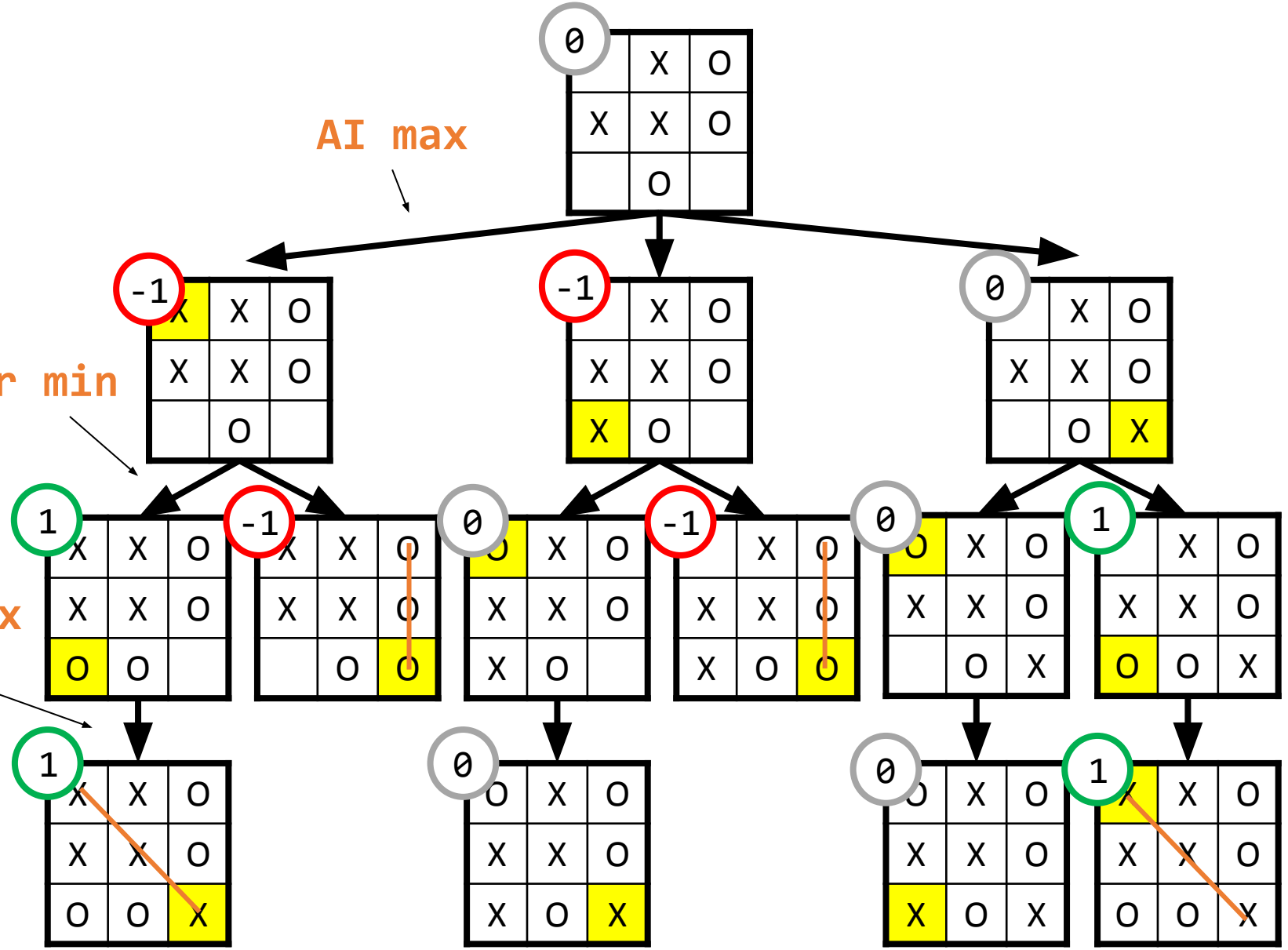
Scoring Game States

How do we score the intermediate states? Look at the scores of the state's children.

If the next move is made by the agent, take the **maximum** of the scores.

If it's made by the opponent, take the **minimum**.

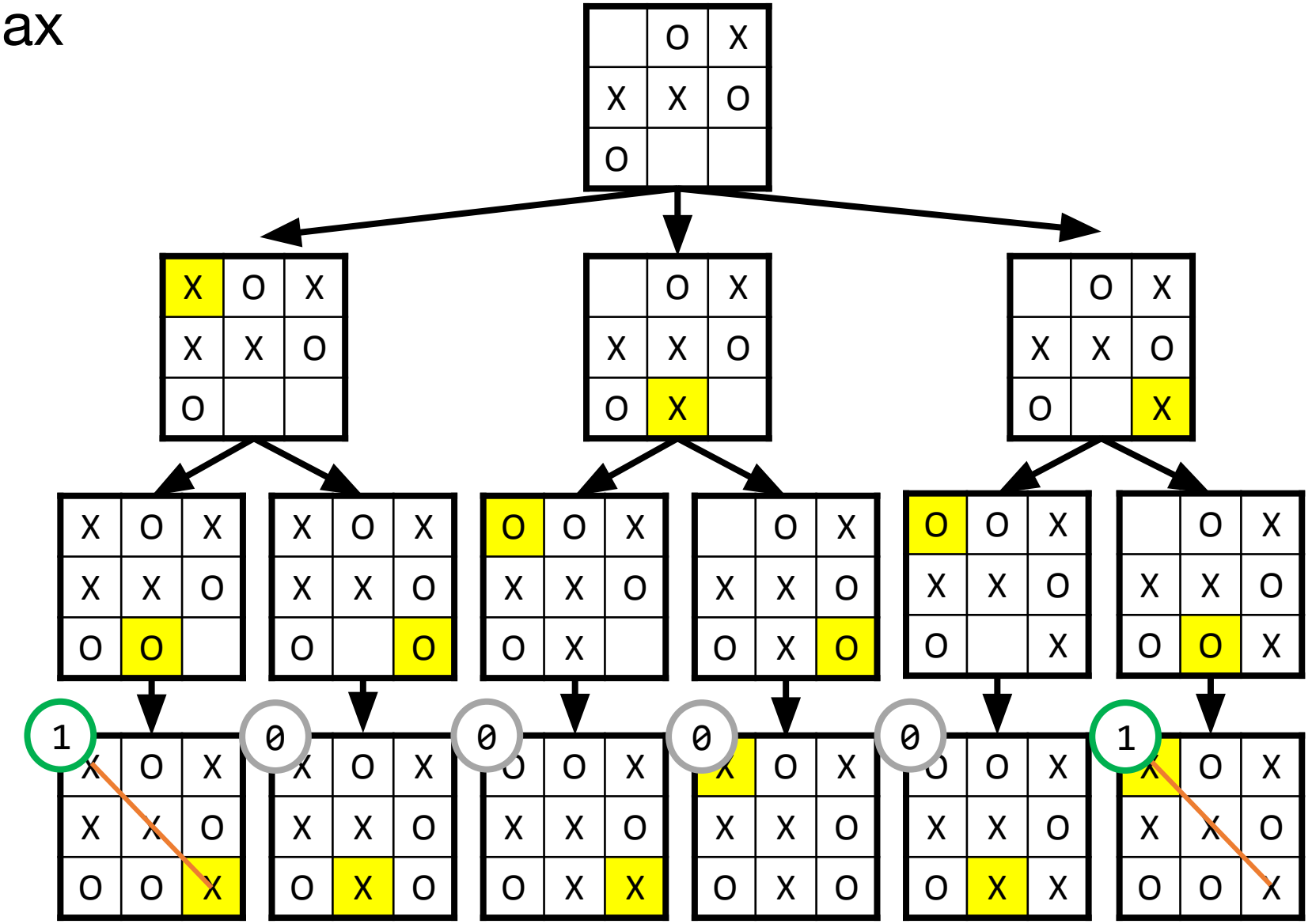
Start from the leaves and build up to the root.



Activity: Apply Minimax

You do: given the tree to the right, apply minimax to find the score of the root node.

Note that the first action is taken by the AI agent.



Minimax Algorithm

```
# Need to use a general tree- "children" instead of "left" and "right"
def minimax(tree, isMyTurn):
    if len(tree["children"]) == 0:
        return score(tree["contents"]) # base case: score of the leaf
    else:
        results = [] # recursive case: get scores of all children
        for child in tree["children"]:
            # switch whose turn it will be for the children
            results.append(minimax(child, not isMyTurn))
        if isMyTurn == True:
            return max(results) # my turn? maximize!
        else:
            return min(results) # opponent's turn? minimize!

def score(state):
    ??? # this depends on the agent's goal
```

Complexity of Minimax

How efficient is minimax? It needs to visit **every node** of the tree, so if the tree has n nodes, it runs in O(n) time.

However, complete game trees are **huge;** more complex games have much larger trees. For example, in Chess there's an average of 35 possible next moves per turn, with an average of 100 turns per game. That means there are 35^{100} possible states to check – way too many!!

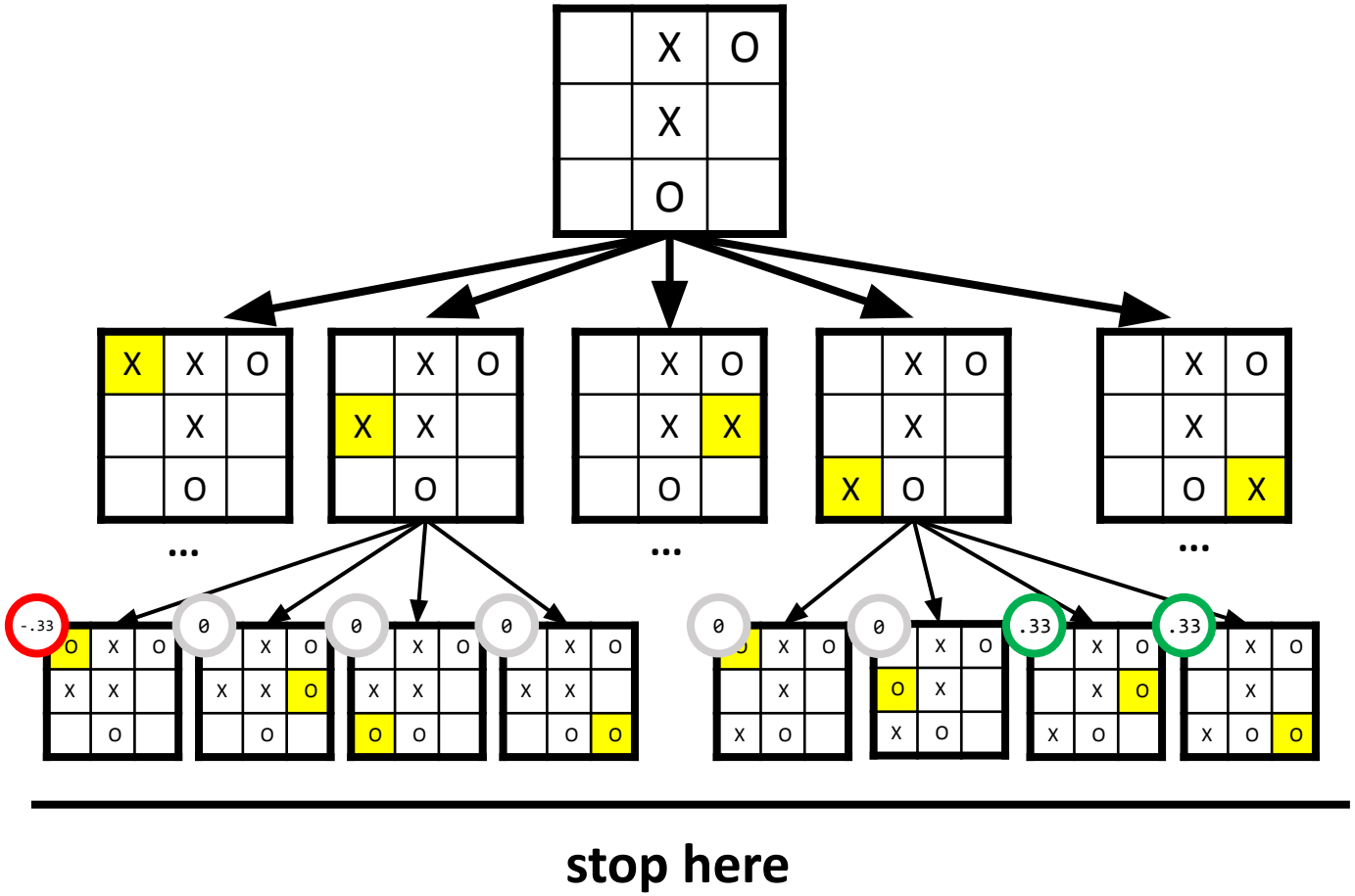
In general, AI agents will try to constrain the size of a game tree by using **heuristics**, as we discussed in the Tractability lecture.

Heuristics in Minimax

The main flaw in minimax is the size of the game tree. We can address this by having the computer move down a set number of levels in the game tree, then stop, even if it has not reached an end state.

For states that are not leaves, use a heuristic to **score** the state based on the current setup of the game. Then the agent can use minimax to find the next-best move based on the heuristic scores.

If the heuristic is well-designed, its score should approximate the real result and minimax should still produce a good result!



Heuristic:

(number of possible X wins - number of possible O wins)

---

total number of non-tie results

Sidebar: Game AIs

Algorithms like minimax and the use of heuristics have made it possible for AI agents to beat world champions at games like Chess, Go, and Poker.

Why did it take 19 years to get from Chess to Go? Go has many more next moves than Chess, so it needed more advanced algorithms (including Monte Carlo randomization and machine learning!).

These AI agents will keep improving as computers grow more powerful and we design better algorithms.

DeepBlue beat chess grandmaster Garry Kasparov in 1997

AlphaGo beat 9-dan ranked Go champion Lee Sedol in 2016