**PokerBots Course Notes**

**Florida International University**

**Competition Lecture 4 - Machine Learning 1**

Contents

4.1 What is Machine Learning?

4.2 Foundational Algorithms

4.3 Creating a Good Model

4.4 Reinforcement Learning

4.5 Multi-Agent Scenarios

4.6 Q-Learning

4.7 Bandit PokerBot

**4.3 Machine Learning**

Why use Machine Learning (ML)?

* Using a machine (computation) to find patterns in data
* Found patterns can be applied without explicit programming
* Key idea: generalize instead of memorize
* Typically utilize large amounts of data
* Used for prediction, data compression, automation, decision-making

ML Workflow Overview   
Acquire dataset ● Create model(s) ● Evaluate model(s) ● Hopefully have good model via selection and improvement ● Deploy

Types of Learning

* Supervised Learning
  + Dataset has examples of correct behavior
  + “Questions with answers”
  + Prediction tasks
* Unsupervised Learning
* Dataset has no notion of good behavior
  + Task is to find patterns to represent dataset
  + Clustering, compression, generative modeling
* Reinforcement Learning (RL)
  + Environment with feedback (rewards), but no correct target
  + Sequential decision making, control, pokerbots!

Problem Setting

Example:

* Supervised learning setting (which we will focus on today)
* We love cats and birds!
* We want our algorithm to take a photo and tell us if it’s a cat or a bird
* We have example photos and labels of both

Key Variables in Supervised Learning

* data example contains
  + “input” properties called features
  + “output” property called label
  + dataset is a collection of these examples
  + an algorithm that takes as an input an incomplete example with only features, and outputs a label to complete the example.

Types of Prediction Classification

* Prediction Example: “cat or bird”
* Regression:
* Predict continuous value aka real number(s) Example: giveaway estimations

**Foundational Algorithms**

Linear Regression

Linear relation between input and output

* “Line of best fit”
* y = w·x + b
* Can generalize to multiple variables
  + y = w1 ·x1 + w2 ·x2 + ··· + b
* Easy to work with but only works well under certain conditions

Logistic Regression

Modifies linear regression for binary classification

* Sigmoid function σ(x) transforms any real number onto interval (0,1).
* y = σ(w·x + b)
* Interpreted as outputting a probability of belonging to a specified class

k-Nearest Neighbors (kNN)

Classification or regression

* Decision using proximity
* For an input:
  + Find the k closest neighbors within dataset
  + Aggregate their labels to decide output
    - Majority vote for classification
    - Average for regression
* Very flexible and can represent complex patterns
* Struggles with higher dimensions

**Creating a Good Model**

What is a model?

* Template for an algorithm
* Model class or hypothesis class is the collection of all possible algorithms that follow that template
* Some models are defined by parameters
  + Ex: w and b in linear regression
  + Space of possible parameters ↔ model class
  + Choosing a model equates to choosing the parameters
* Other models are non-parametric (kNN)
* The larger the model class, the more expressive a model is

Loss Functions

Metric for seeing if model is good by measuring the error it makes on an example

This equates to measuring some distance between predicted and true label

* Linear Regression: squared distance
* Logistic Regression: cross-entropy
* kNN: 0/1 accuracy rate

Loss function reports a penalty for a single input-output example: model is evaluates based on cumulative or average loss across whole set of examples

Combine everything together!

Consider all models within the hypothesis class and pick the one that has the best (lowest) loss across the entire dataset

For parametric models, this is a minimization problem over the parameter space

* “Training”
* Sometimes near impossible to find the absolute best one so approximations are settled for

Why don’t we simply just use the most expressive hypothesis class? More options means that the best options are going to have better results…

* Ex: the space of polynomial functions can do more than everything linear functions can

Overfitting

“Too good to be true”

* Model fits dataset extremely well but fails to capture underlying relationship (which is what we want)
* Memorization occurs, which prevents generalization
* Caused by overexpressive hypothesis class, aka too many parameters, relative to number of data points

Model Evaluation

Split dataset into training and testing groups

* Create model only using train set
* Evaluate model based on loss across test set

Goal is to detect overfitting and have a more realistic measure of a good model

Perform the split randomly

* Each data point has equal chance of showing up in either
* Idea is to have the data be from the same distribution but still different

Analogy: Practice exams vs Real exam

* Test same capabilities using different questions
* If real exam was exact copy of practice, then doing well doesn’t say much about what was learned

When we deploy model it may fail - we want to assess whether this would happen (ideally not)

Sources of Error Bias

* error due to simplistic assumptions
* model class not expressive/flexible enough to represent data
* consistent error resulting from approach

Characterized by high loss in both training and testing

* Ex: Linear Regression Variance
* error from sensitivity to training data

Model class is overly flexible, attempting to capture noise as a pattern

Characterized by low loss in training but high loss in testing (aka overfitting)

* Ex: kNN

Bias-Variance Tradeoff

There is an inherent tradeoff between bias and variance within model selection

Many design decisions (aka hyperparameters) increase/decrease the model’s complexity which could lead to too much bias or variance

Some considerations to find the “sweet spot”:

* Validation sets ○ Regularization
* Careful feature selection/pruning

**4.4 Reinforcement Learning**

Training an agent to make decisions by interacting with an environment to maximize some reward signal aggregated over many steps. The task usually involves sequentially taking actions and observing their effects.

Examples:

* Most board/card/video games
* Control: robotics, self-driving cars
* Recommendation: advertisements
* Agent settings: chatbots

Contrast with other types of Learning

Supervised and unsupervised learning analyze a fixed dataset and seek a model that closely aligns with that dataset.

Reinforcement learning works within an environment and seek a behavior that performs well The data is the history of interactions with the environment

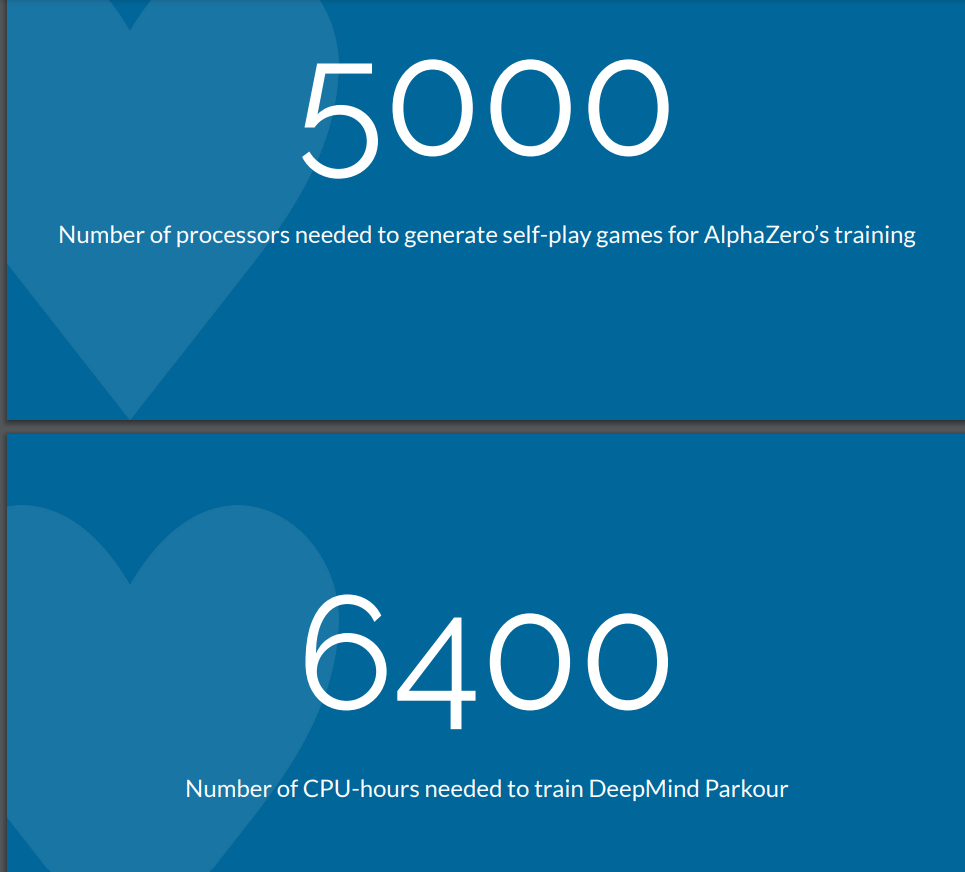
Not necessarily fixed if more interactions can be made, sometimes deciding how to interact in order to collect more data is the hard part

Sometimes the environment itself still needs to be studied, as this can help with navigating to get rewards

Successes

AlphaZero: trained entirely from self-play

* Beat best-in-world chess engine (stockfish) starting from only the rules of the game
* DeepMind “parkour” paper:
  + Inputs: terrain map, joint angles, angular velocities
  + Reward: forward progress



Struggles with multi-agent scenarios

In chess, if bot1 loses to bot2 and bot2 loses to bot3, then there is a good chance that bot3 is the strongest chess player

Gives a clear route to iterated improvement with techniques such as self-play

This is far from guaranteed in poker

Game theoretic settings don’t have the luxuries of supervised learning

Return to rock-paper-scissors

When two reinforcement learning agents are trained against each other, they get very good at beating each other

Risk of getting caught in a cycle without making meaningful improvements to performance

A: rock ⟶ B: paper ⟶ A: scissors ⟶ B: rock ⟶ A: paper ⟶ B: scissors ⟶ A: rock…

Cycles are predictable, which is undesirable

What to reward?

Designing a good reward function is hard

Luckily in poker, the job is done for us

Specification gaming

Specification Gaming

Example: ChatGPT

Reinforcement Learning from Human Feedback (RLHF)

Suppose we’ve considered all the warnings…

* Well-crafted reward function
* Clear goal in mind for what we want
* Sufficient compute resources
* Handling multi-agent scenarios
* How do we train our policy?

Formalization

Problem Setting

Agent: Learner/decision-maker.

Environment: World the agent interacts with.

State: Current situation.

Action: Decision made by the agent.

Policy: How our agent decides which action to take.

Reward: Feedback for the action.

Trajectory: (initial) state

(chosen by agent’s policy)

Reward (environment

Next state (environment)

Action…

Problem Setting

State Space S

Action Space A

Reward Function R(s,a)

Transition Function P(s’ | s,a)

Objective Expected sum of rewards

Multi-armed Bandit

Simpler Example:

The ‘game’ only has one state, with multiple fixed actions

Each action has its own reward, potentially drawn from some distribution

We have to choose one action to pick that gives us the most (expected) reward, but we can play many times

Examples:

* Recommendation feeds (tiktok)
* Choosing a restaurant to eat at

Approaching the Bandit Problem

We would like to figure out which actions are the most fruitful (long term gain), while also using our turns to take actions that we already know are fruitful (short term gain)

This tradeoff between exploration and exploitation is a general tradeoff that appears all throughout RL

Biphasic Approach ○ randomly choose for some predetermined amount of steps ○ afterwards only pick the empirical best

Epsilon-greedy

With some small probability, take a random action

Otherwise pick the action with the best average history of rewards

Bandits in Pokerbots:

Learning across hands

We have a collection of bots with different behaviors

At each hand (or run of 25 hands), we pick a bot to use to decide the action

Leads to some random reward (chip delta)

Finding the bot that does well against our particular opponent across hands can help us counter them with an exploitative strategy

This can be good for building on existing bots that don’t have an obvious one that’s better, and don’t carry state variables across rounds/runs

Q-learning

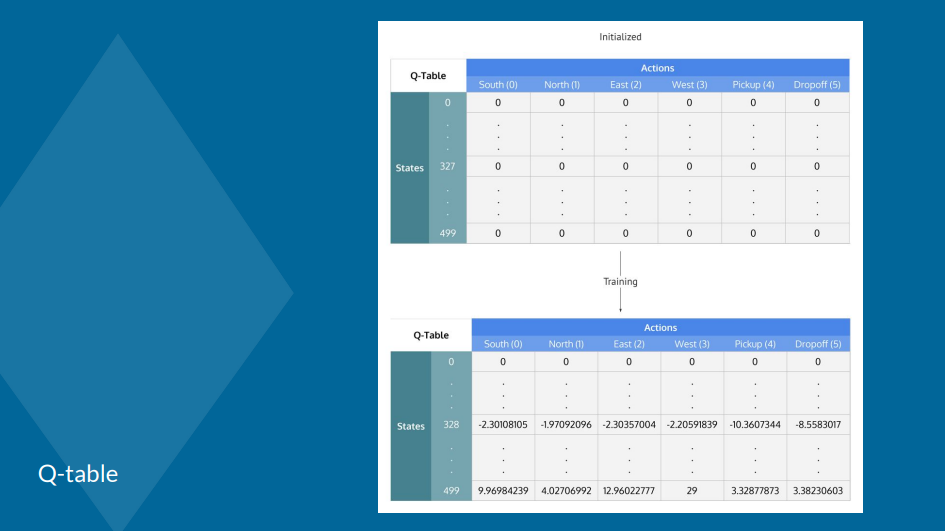
Let’s return to thinking of poker as a multi-step process

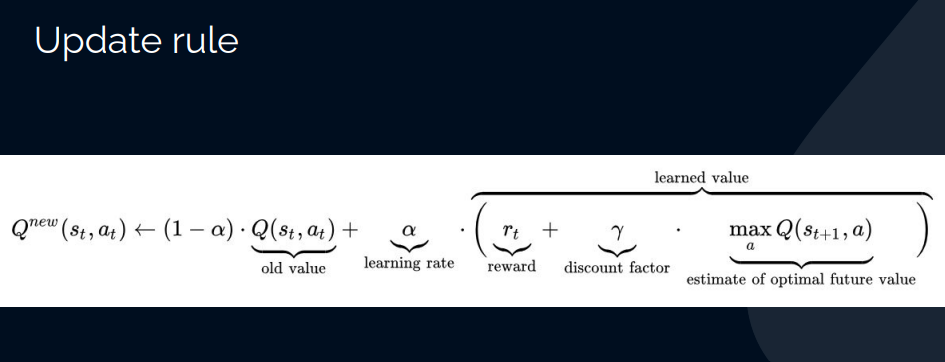
* Extensive form instead of matrix form in the game theory lecture

Want to find a value function Q(s, a) for each state-action pair representing how good it is to be in that state taking that action

If it’s accurate, then we simply choose the action that has the highest stored value!

Q-learning is a sample-based, one-agent way to tabulate the game states of this process and the quality of each action we could take in any given game state





The upsides of Q-learning

General: Q-learning can learn a policy to maximize final reward even if rewards happen incrementally

Simple and intuitive: we repeatedly play games, and we take the deterministic action with the highest quality (do what worked well in the past)

Theoretically sound: for any finite Markov decision process, Q-learning finds a policy that maximizes expected reward

More downsides

Model-based

Slow to converge

Prone to getting stuck in local optima

Intractable if the state-action space is too large

What do we do?

* Use neural networks as an approximation (DQN)
* Use a specialized algorithm for large, imperfect information games

Value Iteration

Instead of a function for every state-action pair, seek a function V(s) that reports the value of being in a given state

To choose actions, we try to pick one that leads to the highest next state (or average if nondeterministic transitions)

This can save compared to Q-learning if action space is large

However it still faces many of the other downsides in Q-learning

Policy Methods Direct Policy Optimization:

Our behavior is directly governed by a policy π(a|s) that tells us how to (possibly randomly) choose an action from a given state.

* Player.get\_action
* If we believe there’s some idea of correct actions, then we can apply supervised approaches Actor-Critic Methods:
* Actor: Updates policy.
* Critic: Evaluates policy using value function.

Idea for Bandit Bot At the beginning of every match:

Initialize which bots we’ll use At new round:

Pick which bot to use for the round At each action:

Use the stored bot At end of round:

Update statistics based on delta