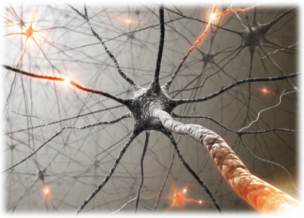
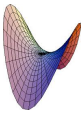
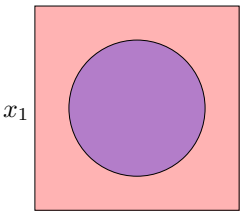
Lecture 6.1: Machine learning V





Review: Features



$$\phi(x) = [1, x_1, x_2, x_1^2, x_2^2]$$

Review: Features

Regression: $x \in \mathbb{R}, y \in \mathbb{R}$

Piecewise constant functions:

$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \dots]$$

$$\mathcal{F}_3 = \{x \mapsto \textstyle\sum_{j=1}^{10} w_j \mathbf{1}[j-1 < x \leq j] : \mathbf{w} \in \mathbb{R}^{10}\}$$

Roadmap

Neural networks

Backpropagation

Nearest neighbors

Summary

- What we've shown so far is that by being mildly clever with choosing the feature extractor ϕ, we can actually get quite a bit of mileage out of our so-called linear predictors.
- However, sometimes we don't know what features are good to use, either because the prediction task is non-intuitive or we don't have time to figure out which features are suitable. Sometimes, we think we might know what features are good, but then it turns out that they aren't (this happens a lot!).
- In the spirit of machine learning, we'd like to automate things as much as possible. In this context, it means creating algorithms that can take whatever crude features we have and turn them into refined predictions, thereby shifting the burden off feature extraction and moving it to learning.
- Neural networks have been around for many decades, but they fell out of favor because they were difficult to train. In the last decade, there has been a huge resurgence of interest in neural networks since they perform so well and training seems to not be such an issue when you have tons of data and compute.
- In a way, neural networks allow one to automatically learn the features of a linear classifier which are geared towards the desired task, rather than specifying them all by hand.

Motivation

Example: predicting car collision

Input: position of two oncoming cars $x = [x_1, x_2]$

Output: whether safe ($y = +1$) or collide ($y = -1$)

True function: safe if cars sufficiently far

$$y = \text{sign}(|x_1 - x_2| - 1)$$

Examples:

x	y
$[1, 3]$	$+1$
$[3, 1]$	$+1$
$[1, 0.5]$	-1

- As a motivating example, consider the problem of predicting whether two cars at positions x_1 and x_2 are going to collide. Suppose the true output is 1 (safe) whenever the cars are separated by a distance of at least 1. Clearly, this decision boundary is not linear.

Decomposing the problem

Test if car 1 is far right of car 2:

$$h_1 = \mathbf{1}[x_1 - x_2 \geq 1]$$

Test if car 2 is far right of car 1:

$$h_2 = \mathbf{1}[x_2 - x_1 \geq 1]$$

Safe if at least one is true:

$$y = \text{sign}(h_1 + h_2)$$

x	h_1	h_2	y
$[1, 3]$	0	1	$+1$
$[3, 1]$	1	0	$+1$
$[1, 0.5]$	0	0	-1

- The intuition is to break up the problem into two subproblems, which test if car 1 (car 2) is to the far right.
- Given these two binary values h_1, h_2, we can declare safety if at least one of them is true.

Learning strategy

Define: $\phi(x) = [1, x_1, x_2]$

Intermediate hidden subproblems:

$$h_1 = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_1 = [-1, +1, -1]$$

$$h_2 = \mathbf{1}[\mathbf{v}_2 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_2 = [-1, -1, +1]$$

Final prediction:

$$f_{\mathbf{V},\mathbf{w}}(x) = \text{sign}(w_1 h_1 + w_2 h_2) \qquad \mathbf{w} = [1, 1]$$

Key idea: joint learning

Goal: learn both hidden subproblems $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2)$ and combination weights $\mathbf{w} = [w_1, w_2]$

- Having written y in a specific way, let us try to generalize to a family of predictors (this seems to be a recurring theme).
- We can define $\mathbf{v}_1 = [-1, 1, -1]$ and $\mathbf{v}_2 = [-1, -1, 1]$ and $w_1 = w_2 = 1$ to accomplish this.
- At a high-level, we have defined two intermediate subproblems, that of predicting h_1 and h_2. These two values are hidden in the sense that they are not specified to be anything. They just need to be set in a way such that y is linearly predictable from them.

Gradients

Problem: gradient of h_1 with respect to $\mathbf{v}_1$ is 0

$$h_1 = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$

[whiteboard]

Definition: logistic function

The logistic function maps $(-\infty, \infty)$ to $[0, 1]$:

$$\sigma(z) = (1 + e^{-z})^{-1}$$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

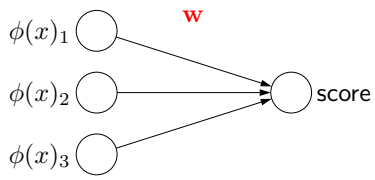
Solution:

$$h_1 = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

- If we try to train the weights $\mathbf{v}_1, \mathbf{v}_2, w_1, w_2$, we will immediately notice a problem: the gradient of h_1 with respect to $\mathbf{v}_1$ is always zero because of the hard thresholding function.
- Therefore, we define a function **logistic function** $\sigma(z)$, which looks roughly like the step function $\mathbf{1}[z \geq 0]$, but has non-zero gradients everywhere.
- One thing to bear in mind is that even though the gradients are non-zero, they can be quite small when $|z|$ is large. This is one reason why optimizing neural networks is hard.

Linear predictors

Linear predictor:



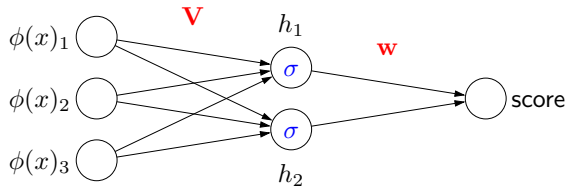
Output:

$$\text{score} = \mathbf{w} \cdot \phi(x)$$

- Let's try to visualize the predictors.
- Recall that linear classifiers take the input $\phi(x) \in \mathbb{R}^d$ and directly take the dot product with the weight vector $\mathbf{w}$ to form the score, the basis for prediction in both binary classification and regression.

Neural networks

Neural network:



Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

- The idea in neural networks is to map an input $\phi(x) \in \mathbb{R}^d$ onto a hidden **intermediate representation** $\mathbf{h} \in \mathbb{R}^k$, which in turn is mapped to the score.
- Specifically, let k be the number of hidden units. For each hidden unit $j = 1, \ldots, k$, we have a weight vector $\mathbf{v}_j \in \mathbb{R}^d$, which is used to determine the value of the hidden node $h_j \in \mathbb{R}$ (also called the **activation**) according to $h_j = \sigma(\mathbf{v}_j \cdot \phi(x))$, where σ is the activation function. The activation function can be a number of different things, but its main property is that it is a non-linear function. Traditionally the **sigmoid** function $\sigma(z) = (1 + e^{-z})^{-1}$ was used, but recently the **rectified linear** function $\sigma(z) = \max\{z, 0\}$ has gained popularity.
- Let $\mathbf{h} = [h_1, \ldots, h_k]$ be the vector of activations. This activation vector is now combined with another weight vector $\mathbf{w} \in \mathbb{R}^k$ to produce the final score.

Neural networks

Interpretation: intermediate hidden units as learned features of a linear predictor

Key idea: feature learning

Before: apply linear predictor on manually specify features

$$\phi(x)$$

Now: apply linear predictor on automatically learned features

$$h(x) = [h_1(x), \ldots, h_k(x)]$$

- The noteworthy aspect here is that the activation vector $\mathbf{h}$ behaves a lot like our feature vector $\phi(x)$ that we were using for linear prediction. The difference is that mapping from input $\phi(x)$ to $\mathbf{h}$ is learned automatically, not manually constructed (as was the case before). Therefore, a neural network can be viewed as learning the features of a linear classifier. Of course, the type of features that can be learned must be of the form $x \to \sigma(\mathbf{v}_j \cdot \phi(x))$.
- Whether this is a suitable form depends on the nature of the application. Empirically, though, neural networks have been quite successful, since learning the features from the data with the explicit objective of minimizing the loss can yield better features than ones which are manually crafted. Recently, there have been some advances in getting neural networks to work, and they have become the state-of-the-art in many tasks. For example, all the major companies (Google, Microsoft, IBM) all recently switched over to using neural networks for speech recognition. In computer vision, (convolutional) neural networks are completely dominant in object recognition.

Roadmap

Neural networks

Backpropagation

Nearest neighbors

Summary

Motivation: loss minimization

Optimization problem:

$$\min_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V},\mathbf{w})$$

$$\text{TrainLoss}(\mathbf{V},\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x,y,\mathbf{V},\mathbf{w})$$

$$\text{Loss}(x,y,\mathbf{V},\mathbf{w}) = (y - f_{\mathbf{V},\mathbf{w}}(x))^2$$

$$f_{\mathbf{V},\mathbf{w}}(x) = \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x))$$

Goal: compute gradient

$$\nabla_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V},\mathbf{w})$$

- The main thing left to do for neural networks is to be able to train them. Conceptually, this should be straightforward: just take the gradient and run SGD.
- While this is true, computing the gradient, even though it is not hard, can be quite tedious to do by hand.

Approach

Mathematically: just grind through the chain rule

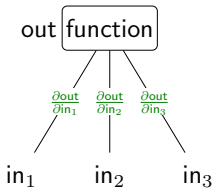
Next: visualize the computation using a computation graph

Advantages:

- Avoid long equations
- Reveal structure of computations (modularity, efficiency, dependencies)

- We will illustrate a graphical way of organizing the computation of gradients, which is built out of a few components.
- This graphical approach will show the structure of the function and will not only make gradients easy to compute, but also shed more light onto the predictor and loss function.
- In fact, these days if you use a package such as TensorFlow or Pytorch, you can write down the expressions symbolically and the gradient is computed for you. This is done essentially using the computational procedure that we will see.

Functions as boxes



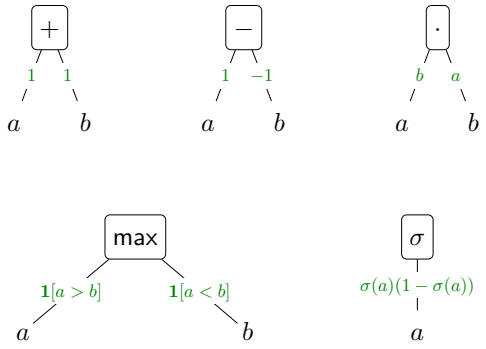
Partial derivatives (gradients): how much does the output change if an input changes?

Example:

$2\text{in}_1 + (\text{in}_2 + \epsilon)\text{in}_3 = \text{out} + \text{in}_3\epsilon$

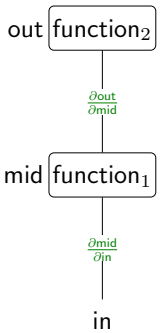
- The first conceptual step is to think of functions as boxes that take a set of inputs and produces an output. Then the partial derivatives (gradients if the input is vector-valued) are just a measure of sensitivity: if we perturb in_1 by a small amount ϵ, how much does the output out change? The answer is $\frac{\partial \text{out}}{\partial \text{in}_1} \cdot \epsilon$. For convenience, we write the partial derivative on the edge connecting the input to the output.

Basic building blocks



- Here are 5 examples of simple functions and their partial derivatives. These should be familiar from basic calculus. All we've done is to present them in a visually more intuitive way.
- But it turns out that these simple functions are all we need to build up many of the more complex and potentially scarier looking functions that we'll encounter in machine learning.

Composing functions



Chain rule:

$$\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$$

- The second conceptual point is to think about **composing**. Graphically, this is very natural: the output of one function f simply gets fed as the input into another function g.
- Now how does in affect out (what is the partial derivative)? The key idea is that the partial derivative **decomposes** into a product of the two partial derivatives on the two edges. You should recognize this is no more than the chain rule in graphical form.
- More generally, if the partial derivative of y with respect to x is simply the product of all the green expressions on the edges of the path connecting x and y. This visual intuition will help us better understand more complex functions, which we will turn to next.

Binary classification with hinge loss

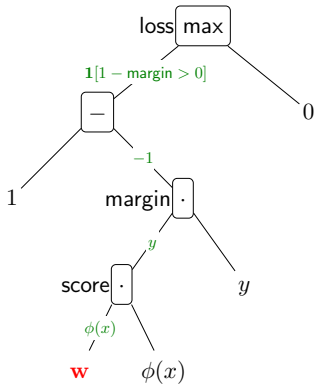
Hinge loss:

$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x)y, 0\}$$

Compute:

$$\frac{\partial \text{Loss}(x,y,\mathbf{w})}{\partial \mathbf{w}}$$

Binary classification with hinge loss



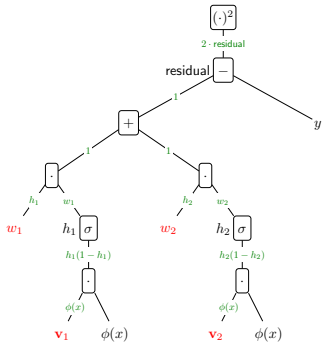
Gradient: multiply the edges

$$-\mathbf{1}[\text{margin} < 1]\phi(x)y$$

- Let us start with a simple example: the hinge loss for binary classification.
- In red, we have highlighted the weights $\mathbf{w}$ with respect to which we want to take the derivative. The central question is how small perturbations in $\mathbf{w}$ affect a change in the output (loss). Intermediate nodes have been labeled with interpretable names (score, margin).
- The actual gradient is the product of the edge-wise gradients from $\mathbf{w}$ to the loss output.

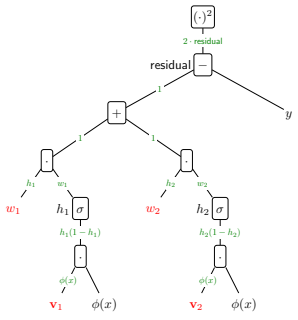
Neural network

$$\text{Loss}(x, y, \mathbf{w}) = \left(\sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y\right)^2$$



- Now, we can apply the same strategy to neural networks. Here we are using the squared loss for concreteness, but one can also use the logistic or hinge losses.
- Note that there is some really nice modularity here: you can pick any predictor (linear or neural network) to drive the score, and the score can be fed into any loss function (squared, hinge, etc.).

Backpropagation



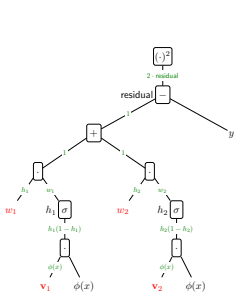
Definition: Forward/backward values

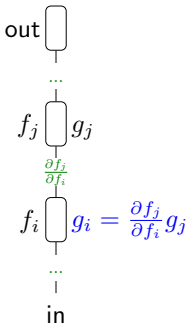
Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{out}}{\partial f_i}$ is how f_i influences output

- So far, we have mainly used the graphical representation to visualize the computation of function values and gradients for our conceptual understanding. But it turns out that the graph has algorithmic implications too.
- Recall that to train any sort of model using (stochastic) gradient descent, we need to compute the gradient of the loss (top output node) with respect to the weights (leaf nodes highlighted in red).
- We also saw that these gradients (partial derivatives) are just the product of the local derivatives (green stuff) along the path from a leaf to a root. So we can just go ahead and compute these gradients: for each red node, multiply the quantities on the edges. However, notice that many of the paths share subpaths in common, so sometimes there's an opportunity to save computation (think dynamic programming).
- To make this sharing more explicit, for each node i in the tree, define the forward value f_i to be the value of the subexpression rooted at that tree, which depends on the inputs underneath that subtree. For example, the parent node of w_1 corresponds to the expression $w_1\sigma(\mathbf{v}_1 \cdot \phi(x))$. The f_i's are the intermediate computations required to even evaluate the function at the root.
- Next, for each node i in the tree, define the backward value g_i to be the gradient of the output with respect to f_i, the forward value of node i. This measures the change that would happen in the output (root node) induced by changes to f_i.
- Note that both f_i and g_i can either be scalars, vectors, or matrices, but have the same dimensionality.

Backpropagation





Algorithm: backpropagation

Forward pass: compute each f_i (from leaves to root)

Backward pass: compute each g_i (from root to leaves)

- We now define the backpropagation algorithm on arbitrary computation graphs.
- First, in the forward pass, we go through all the nodes in the computation graph from leaves to the root, and compute f_i, the value of each node i, recursively given the node values of the children of i. These values will be used in the backward pass.
- Next, in the backward pass, we go through all the nodes from the root to the leaves and compute g_i recursively from f_i and g_j, the backward value for the parent of i using the key recurrence $g_i = \frac{\partial f_j}{\partial f_i} g_j$ (just the chain rule).
- In this example, the backward pass gives us the gradient of the output node (the gradient of the loss) with respect to the weights (the red nodes).

Roadmap

Neural networks

Backpropagation

Nearest neighbors

Summary

- Linear predictors were governed by a simple dot product $\mathbf{w} \cdot \phi(x)$. Neural networks chained together these simple primitives to yield something more complex. Now, we will consider **nearest neighbors**, which yields complexity by another mechanism: computing similarities between examples.

Nearest neighbors

Algorithm: nearest neighbors

Training: just store $\mathcal{D}_{\text{train}}$

Predictor $f(x')$:

- Find $(x, y) \in \mathcal{D}_{\text{train}}$ where $\|\phi(x) - \phi(x')\|$ is smallest
- Return y

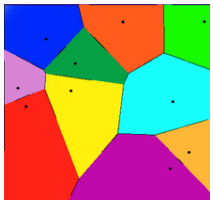
Key idea: similarity

Similar examples tend to have similar outputs.

- **Nearest neighbors** is perhaps conceptually one of the simplest learning algorithms. In a way, there is no learning. At training time, we just store the entire training examples. At prediction time, we get an input x' and we just find the input in our training set that is **most similar**, and return its output.
- In a practical implementation, finding the closest input is non-trivial. Popular choices are using k-d trees or locality-sensitive hashing. We will not worry about this issue.
- The intuition being expressed here is that similar (nearby) points tend to have similar outputs. This is a reasonable assumption in most cases; all else equal, having a body temperature of 37 and 37.1 is probably not going to affect the health prediction by much.

Expressivity of nearest neighbors

Decision boundary: based on Voronoi diagram



- Much more expressive than quadratic features
- **Non-parametric**: the hypothesis class adapts to number of examples
- Simple and powerful, but kind of brute force

- Let's look at the decision boundary of nearest neighbors. The input space is partitioned into regions, such that each region has the same closest point (this is a Voronoi diagram), and each region could get a different output.
- Notice that this decision boundary is much more expressive than what you could get with quadratic features. In particular, one interesting property is that the complexity of the decision boundary adapts to the number of training examples. As we increase the number of training examples, the number of regions will also increase. Such methods are called **non-parametric**.

Summary of learners

- Linear predictors: combine raw features

  prediction is **fast**, learning is **easy**, **weak** use of features

- Neural networks: combine learned features

  prediction is **fast**, learning is **hard**, **powerful** use of features

- Nearest neighbors: predict according to similar examples

  prediction is **slow**, learning is **easy**, **powerful** use of features

- Let us conclude now. First, we discussed some general principles for designing good features for linear predictors. Just with the machinery of linear prediction, we were able to obtain rich predictors which were quite rich.
- Second, we focused on expanding the expressivity of our predictors fixing a particular feature extractor ϕ.
- We covered three algorithms: **linear predictors** combine the features linearly (which is rather weak), but is easy and fast.
- **Neural networks** effectively learn non-linear features, which are then used in a linear way. This is what gives them their power and prediction speed, but they are harder to learn (due to the non-convexity of the objective function).
- **Nearest neighbors** is based on computing similarities with training examples. They are powerful and easy to learn, but are slow to use for prediction because they involve enumerating (or looking up points in) the training data.

Roadmap

Neural networks

Backpropagation

Nearest neighbors

Summary

Summary

- Feature extraction (think hypothesis classes) [modeling]
- Prediction (linear, neural network, k-means) [modeling]
- Loss functions (compute gradients) [modeling]
- Optimization (stochastic gradient, alternating minimization) [algorithms]
- Generalization (think development cycle) [modeling]

- This concludes our tour of the foundations of machine learning, although machine learning will come up again later in the course. You should have gotten more than just a few isolated equations and algorithms. It is really important to think about the overarching principles in a modular way.
- First, feature extraction is where you put your domain knowledge into. In designing features, it's useful to think in terms of the induced **hypothesis classes** — what kind of functions can your learning algorithm potentially learn?
- These features then drive **prediction**: either linearly or through a neural network. We can even think of k-means as trying to predict the data points using the centroids.
- **Loss functions** connect predictions with the actual training examples.
- Note that all of the design decisions up to this point are about modeling. Algorithms are very important, but only come in once we have the right **optimization problem** to solve.
- Finally, machine learning requires a leap of faith. How does optimizing anything at training time help you **generalize** to new unseen examples at test time? Learning can only work when there's a common core that cuts past all the idiosyncrasies of the examples. This is exactly what features are meant to capture.

A brief history

1795: Gauss proposed least squares (astronomy)

1940s: logistic regression (statistics)

1952: Arthur Samuel built program that learned to play checkers (AI)

1957: Rosenblatt invented Perceptron algorithm (like SGD)

1969: Minsky and Papert "killed" machine learning

1980s: neural networks (backpropagation, from 1960s)

1990: interface with optimization/statistics, SVMs

2000s-: structured prediction, revival of neural networks, etc.

- Many of the ideas surrounding fitting functions was known in other fields long before computers, let alone AI.
- When computers arrived on the scene, learning was definitely on people's radar, although this was detached from the theoretical, statistical and optimization foundations.
- In 1969, Minsky and Papert wrote a famous paper *Perceptrons*, which showed the limitations of linear classifiers with the famous XOR example (similar to our car collision example), which killed off this type of research. AI largely turned to rule-based and symbolic methods.
- Since the 1980s, machine learning has increased its role in AI, been placed on a more solid mathematical foundation with its connection with optimization and statistics.
- While there is a lot of optimism today about the potential of machine learning, there are still a lot of unsolved problems.

Challenges

Capabilities:

- More complex prediction problems (translation, generation)
- Unsupervised learning: automatically discover structure

Responsibilities:

- Feedback loops: predictions affect user behavior, which generates data
- Fairness: build classifiers that don't discriminate?
- Privacy: can we pool data together
- Interpretability: can we understand what algorithms are doing?

- Going ahead, one major thrust is to improve the capabilities of machine learning. Broadly construed, machine learning is about learning predictors from some input to some output. The simplest case is when the output is just a label, but increasingly, researchers have been using the same machine learning tools for doing translation (output is a sentence), speech synthesis (output is a waveform), and image generation (output is an image).
- Another important direction is being able to leverage the large amounts of unlabeled data to learn good representations. Can we automatically discover the underlying structure (e.g., a 3D model of the world from videos)? Can we learn a causal model of the world? How can we make sure that the representations we are learning are useful for some other task?
- A second major thrust has to do with the context in which machine learning is now routinely being applied, for example in high-stakes scenarios such as self-driving cars. But machine learning does not exist in a vacuum. When machine learning systems are deployed to real users, it changes user behavior, and since the same systems are being trained on this user-generated data, this results in feedback loops.
- We also want to build ML systems which are fair. The real world is not fair; thus the data generated from it will reflect these discriminatory biases. Can we overcome these biases?
- The strength of machine learning lies in being able to aggregate information across many individuals. However, this appears to require a central organization that collects all this data, which seems poor practice from the point of view of protecting privacy. Can we perform machine learning while protecting individual privacy? For example, local differential privacy mechanisms inject noise into an individual's measurement before sending it to the central server.
- Finally, there is the issue of trust of machine learning systems in high-stakes situations. As these systems become more complex, it becomes harder for humans to "understand" how and why a system is making a particular decision.

Markov decision processes

Key idea: learning

Programs should improve with experience.

Key idea: state-based models

Model the world as transitions between states.

MDPs: state-based models with randomless

Reinforcement learning: learning policies, not predictors

- In the first part of the class, we spent a lot of time on machine learning and search.
- Machine learning is really about programs that can improve with experience.
- In search, we saw the key idea of a state, and we modeled the world in terms of states and actions that took us to other states.
- Next time, we will explore Markov Decision Processes: state-based models in which the result of your actions is influenced by random chance. To act optimally in such an environment, we must *learn* about the different outcomes that can result from each possible action in each possible state. This leads to the idea of reinforcement learning: instead of learning a single predictor that only thinks about one input at a time, we will learn a **policy** that knows how to behave in any state, in order to optimize for both the present and the future.