

Lecture 8: Machine Learning



Gabriel Poesia

Assistant Professor, CSE

University of Michigan, Ann Arbor

poesia@umich.edu

<https://gpoesia.com>

- 0. Administrivia
- 1. Recap and Gibbs sampling
- 2. Learning to predict
- 3. Linear regression
- 4. Generalization

Homework 2 check-in

Project proposal check-in (due Wednesday)

Use office hours if you need feedback on scope, plan, etc (mine today, Jerry's tomorrow)

RECAP

Bayesian networks

A Bayesian network is a DAG plus one conditional distribution per node, $P(X_i \mid Parents(X_i))$.

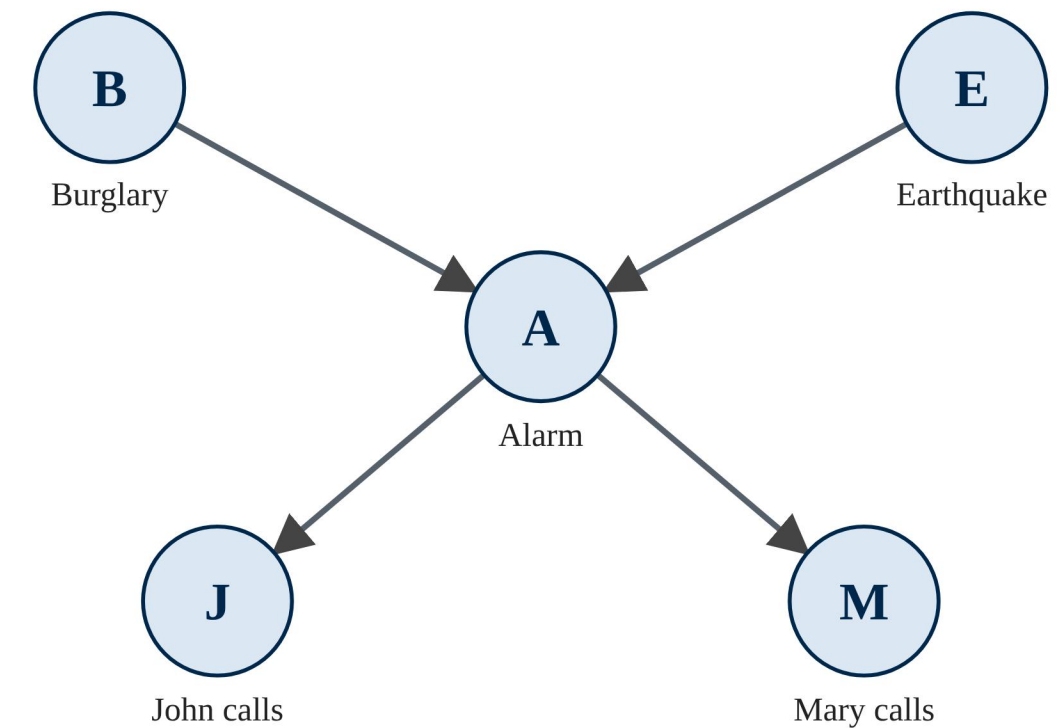
Together, they define one joint distribution:

$$P(x_1, \dots, x_n) = \prod_i P(x_i \mid Parents(X_i))$$

The missing edges encode independence assumptions.

From one model, we can answer many queries, e.g.:

$$P(B \mid j, m) \approx .284$$



Exact inference: variable elimination multiplies factors and sums out hidden variables.

Intermediate factors can get very large. In general, exact inference is #P-hard.

Approximate inference: give up on exact answers, and estimate probabilities from samples.

If we have N samples from a distribution, we can estimate the probability of an event by how often it happens:

$$P(b) \approx \frac{\#\{\text{samples with } B=\text{true}\}}{N}$$

As N grows, frequencies get closer to the true probabilities (law of large numbers).

So if we can sample from a Bayesian Net, we can do inference!

How do we get samples?

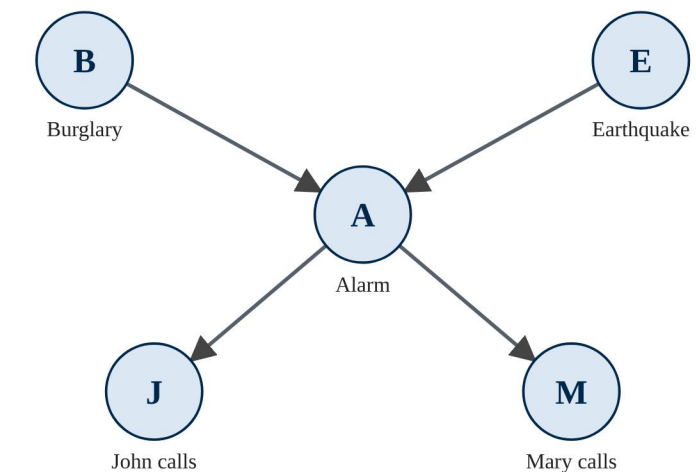
Forward sampling

How do we generate one complete world (realization of random variables) from the network?

Idea (forward sampling): sample in topological order, so parents before children.

At each node, look up the CPT rows that match values sampled for the parents.

To sample a Boolean that is true with probability p : draw r uniformly from $[0, 1]$, and set it to true if $r < p$.



Node	CPT entry	Random r	Value
B	$P(b) = .001$	.42	F
E	$P(e) = .002$	.001	T
A	$P(a \mid \neg b, e) = .29$	.10	T
J	$P(j \mid a) = .90$	.83	T
M	$P(m \mid a) = .70$	.76	F

We get the sample $(\neg b, e, a, j, \neg m)$.

Q: Why is this a sample from the joint?

A: The probability of generating this one assignment is:

$$P(\neg b) P(e) P(a \mid \neg b, e) P(j \mid a) P(\neg m \mid a) \\ = P(\neg b, e, a, j, \neg m)$$

Repeating this gives independent samples from $P(B, E, A, J, M)$.

To estimate probability of a particular assignment just

Rejection sampling

What about a conditional query, like $P(b \mid j, m)$?

Rejection sampling: generate forward samples, and throw away (reject) every sample that disagrees with the evidence.

Sample	B	E	A	J	M	Use?
1	F	F	F	F	F	reject
2	F	T	T	T	F	reject
3	T	F	T	T	T	accept
...						

Among the accepted samples, count how often $B = \text{true}$:

$$P(b \mid j, m) \approx \frac{\#\{\text{samples with } b, j, m\}}{\#\{\text{samples with } j, m\}}$$

The accepted samples are distributed exactly as $P(B, E, A \mid j, m)$. So the estimate is correct in the limit.

But how many samples do we keep? A fraction $P(j, m) \approx .002$ of them: about 2 in every 1000.

To end up with 1000 accepted samples, we need about 480,000 forward samples. With rarer evidence, it gets much worse.

Can we generate only samples that agree with the evidence?

Markov chain Monte Carlo

Idea: keep the evidence fixed, and start with any assignment to the hidden variables.

Then, repeatedly resample query or hidden variables only for many steps

The sequence will get will form a **Markov chain**: stochastic process where the next state only depends on the current (memoryless)

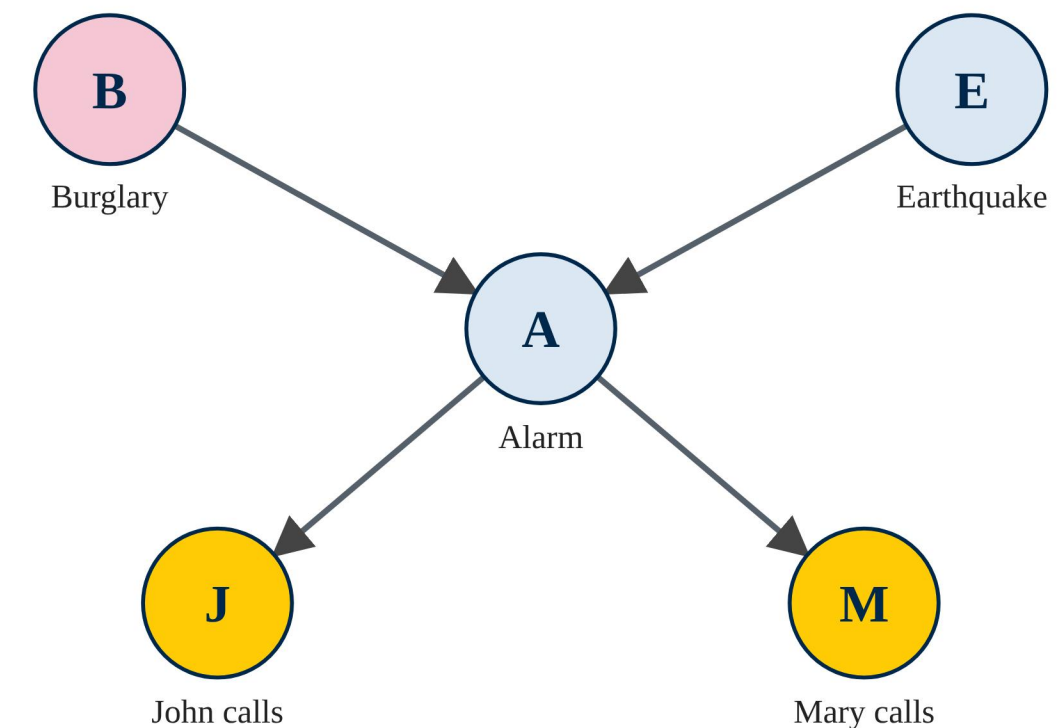
E.g.: with j, m fixed, the chain might visit:

$$(\neg b, \neg e, a) \rightarrow (b, \neg e, a) \rightarrow (b, \neg e, a) \rightarrow (\neg b, \neg e, a) \rightarrow \dots$$

If we choose the transitions well, after enough steps the fraction of time the chain spends in each state approaches the posterior probability of that state.

Then we estimate $P(b \mid j, m)$ by the fraction of visited states with $B = \text{true}$.

Deep theoretical foundations in statistics about what conditions we need for "choosing well"



Query: B Evidence: j, m

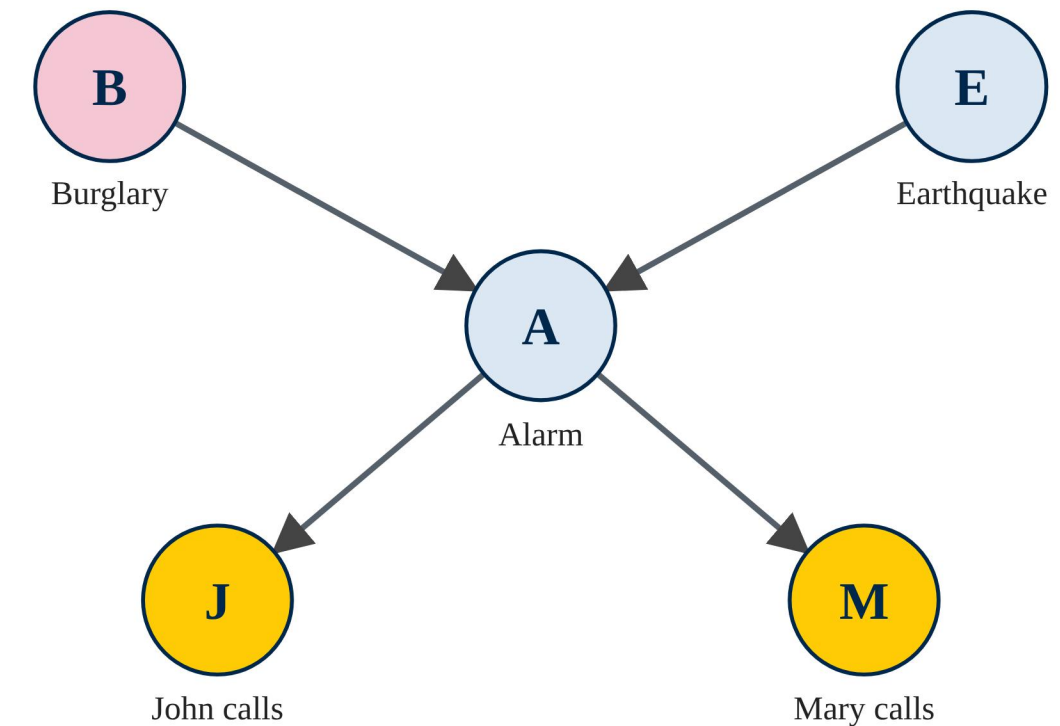


Gibbs sampling is one simple way to choose these transitions:

1. Fix the evidence ($J = M = \text{true}$) and initialize B, E, A arbitrarily.
2. Pick a non-evidence variable X (e.g., cycle through B, E, A).
3. Resample X from $P(X \mid \text{current values of all other variables})$.
4. Repeat. After an initial "burn-in" period, record the states we visit.

Each step changes at most one variable.

But how do we compute $P(X \mid \text{everything else})$?



Markov blanket again

Write x_{-i} for the values of all variables other than X_i . By the definition of conditional probability:

$$P(x_i \mid x_{-i}) = \frac{P(x_i, x_{-i})}{P(x_{-i})}$$

$P(x_{-i})$ doesn't depend on x_i : it's the same for every value of X_i , so it's just the normalization constant.

$$P(x_i \mid x_{-i}) \propto P(x_i, x_{-i}) = P(x_1, \dots, x_n) = \prod_k P(x_k \mid \text{Parents}(X_k))$$

Most factors in the product don't mention X_i either, so they are constants too. Only two kinds are left:

- X_i 's own CPT: $P(x_i \mid \text{Parents}(X_i))$
- the CPT of each child C of X_i : $P(c \mid \text{Parents}(C))$

$$P(x_i \mid x_{-i}) \propto P(x_i \mid \text{Parents}(X_i)) \prod_{C \in \text{Children}(X_i)} P(c \mid \text{Parents}(C))$$

These only involve X_i 's parents, its children, and its children's other parents: its **Markov blanket**!

So each Gibbs update is local, no matter how large the network is.

Gibbs sampling in practice

Consecutive samples are correlated, since each step changes at most one variable.

Also, early samples depend on our arbitrary initial state.

So, in practice we throw away an initial "burn-in" period (discard first K samples)

The chain can also **mix** slowly: take a long time to move between most likely regions.

Example: suppose X and Y agree 99.9% of the time. To go from $X = Y = 0$ to $X = Y = 1$, changing one variable at a time, the chain has to pass through an unlikely state.

Under mild conditions (e.g., all probabilities positive), the visit frequencies do converge to the posterior.

You'll implement Gibbs sampling in Homework 2 to denoise an image.

MCMC is used far beyond Bayesian networks: statistical physics, computational biology, sampling from energy-based models (e.g., to generate images)...

LEARNING TO PREDICT

What did we have to design?

To use a Bayesian network, we had to decide:

- which variables exist (burglary, earthquake, alarm, ...)
- which variables directly depend on which (the graph)
- how to estimate each CPT (priors or from data for each, as in Naïve Bayes)

This gives a model of how the data is generated. Then we used inference to answer questions.

Often, we only care about one kind of question: give some x , what is y ?

email $\rightarrow$ spam?

image $\rightarrow$ which digit?

Now, we'll learn to predict y from x *directly*, from examples, without describing how x and y were generated.

We'll keep some ideas (e.g., fitting parameters to data), but try to avoid strong assumptions about how the data is generated.

Our agent, now

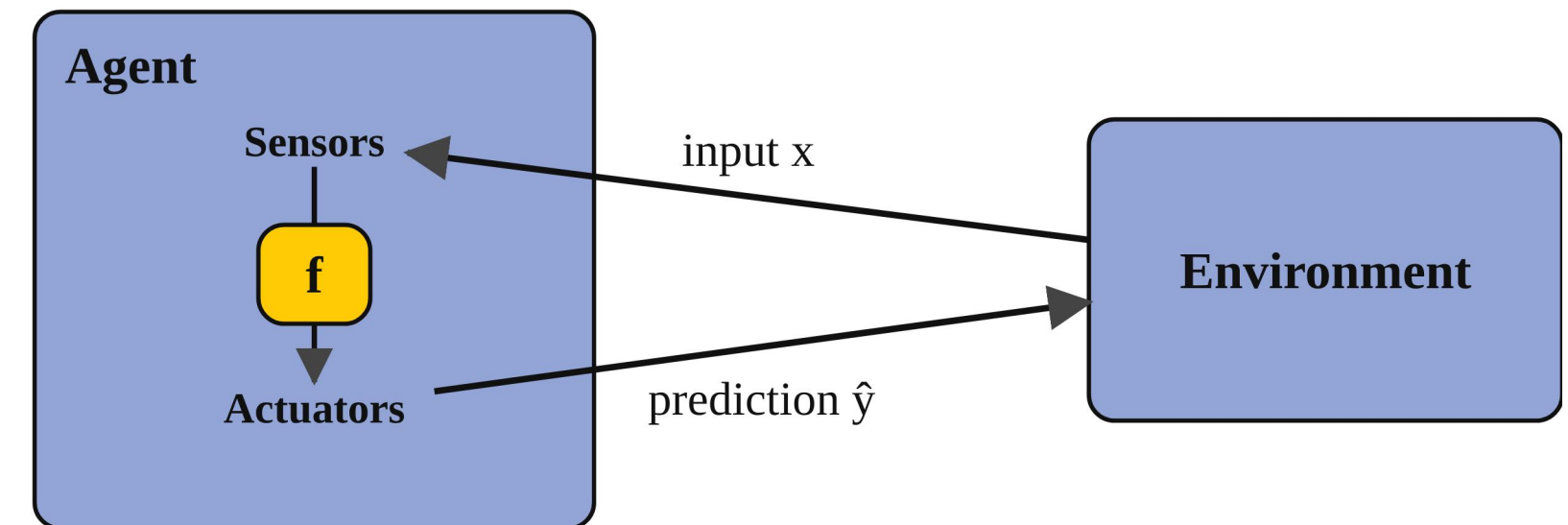
Recall the agent and environment from Lecture 1.

The agent we'll build now is a **predictor** f :

- **Sensors:** an input x (e.g.: email, image, house listing)
- **Action:** make prediction $\hat{y} = f(x)$ (spam or not, a digit, a price)
- **Performance measure:** how wrong its predictions are
- **Environment:** gives sequence of inputs for us to predict, independently (episodic, not sequential)

No planning: the agent maps each percept to an action directly.

f will be learned from **data**: example inputs, with correct output (supervised learning).



Kinds of prediction tasks

The input x can be almost anything. Different kinds of y lead to different tasks:

Task	Output y	Examples
Binary classification	$y \in \{+1, -1\}$	spam or not, fraudulent transaction or not
Multiclass classification	$y \in \{1, \dots, K\}$	which digit is in an image ($K = 10$)
Regression	$y \in \mathbb{R}$	house price, delivery time
Structured prediction	y is a complex object	translation of a sentence, missing edges in a graph, answer to a question

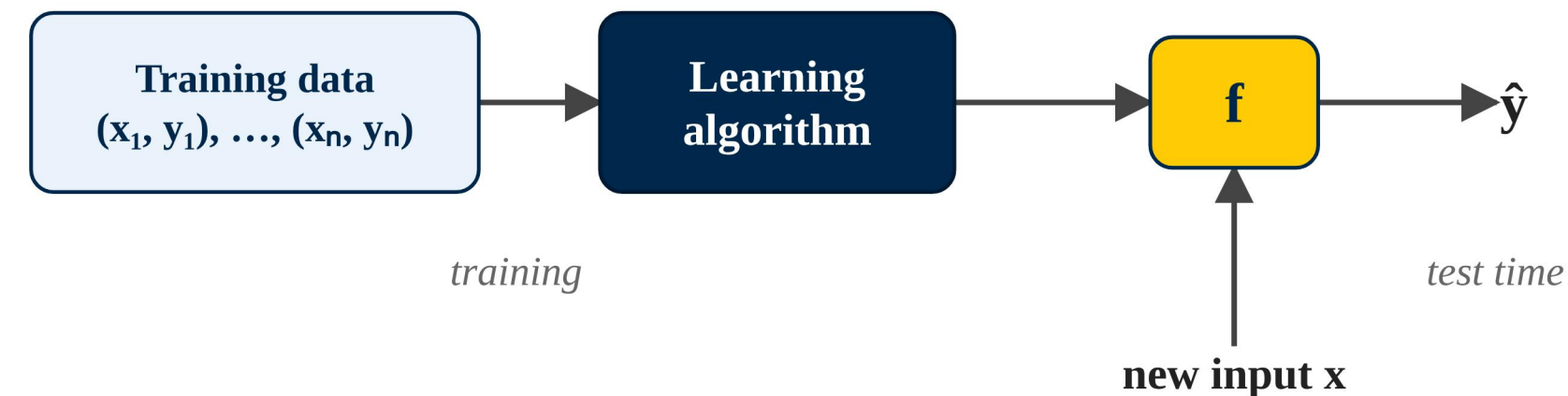
Structured prediction is often reduced to a sequence of classification problems.

E.g., to generate a sentence, predict one word at a time: "The sun rises in the" $\rightarrow$ "east".

This is how LLMs are trained, on the task of "next token prediction". We'll get there soon.

Supervised learning

First paradigm we will study assumes we have labeled training examples.



DEFINITION (SUPERVISED LEARNING)

Given **training data** $\mathcal{D}_{\text{train}} = \{(x_1, y_1), \dots, (x_n, y_n)\}$, a learning algorithm outputs a predictor f . At test time, we only see a new input x , and predict $\hat{y} = f(x)$.

(When the labels are derived from unsupervised data itself, this is often called *self-supervised* learning - like next-token prediction.)

LINEAR REGRESSION

Where is Ceres?

January 1st, 1801: Giuseppe Piazzi finds a new object in the sky, later named Ceres.

He tracks it for about six weeks, until it gets too close to the Sun to be seen.

The question for astronomers in 1801: where will Ceres show up again?

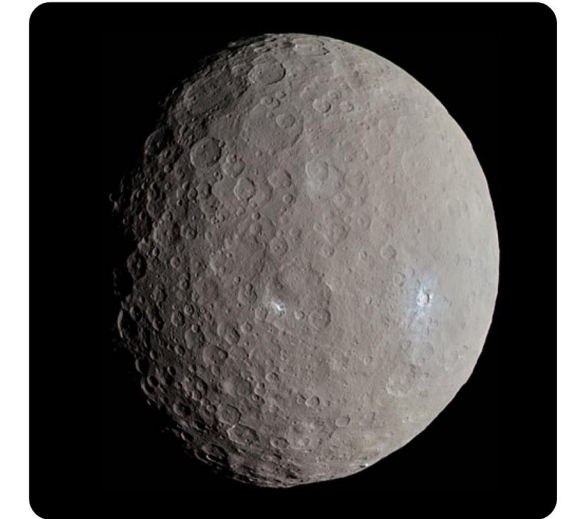
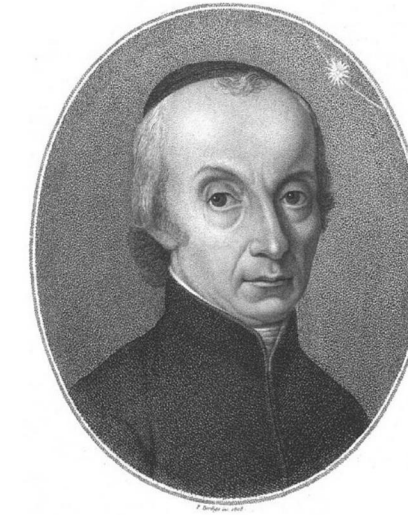
A 24-year-old Carl Friedrich Gauss uses Piazzi's observations to work out the orbit and make a prediction.

In December 1801, Ceres is found again, very close to where Gauss predicted.

His method: choose the model that minimizes the sum of squared errors on the observations, or **least squares**.

(Legendre published it first, in 1805. Gauss published in 1809, saying he had used it since 1795.)

Still one of the most used methods in science, and it already has most of the ingredients of machine learning.



Three design decisions

We'll describe learning methods by answering three questions:

Hypothesis class

Which predictors f can we choose from?

Loss function

Given the training data, how good is a predictor f ?

Optimization algorithm

How do we find the best predictor in our class?

The first two specify the problem: *what* we want. The third is *how* we get it.

We've done this separation before:

	Problem specification	Algorithms
Search	states, actions, costs, goal test	UCS, A*
CSPs	variables, domains, constraints	backtracking, AC-3
Logic	knowledge base, query	resolution, DPLL
Bayesian networks	graph, CPTs, query	variable elimination, Gibbs sampling
Learning	hypothesis class, loss	gradient descent, SGD, AdamW

Same advantage: one optimization algorithm will work for many hypothesis classes and losses.

Training data

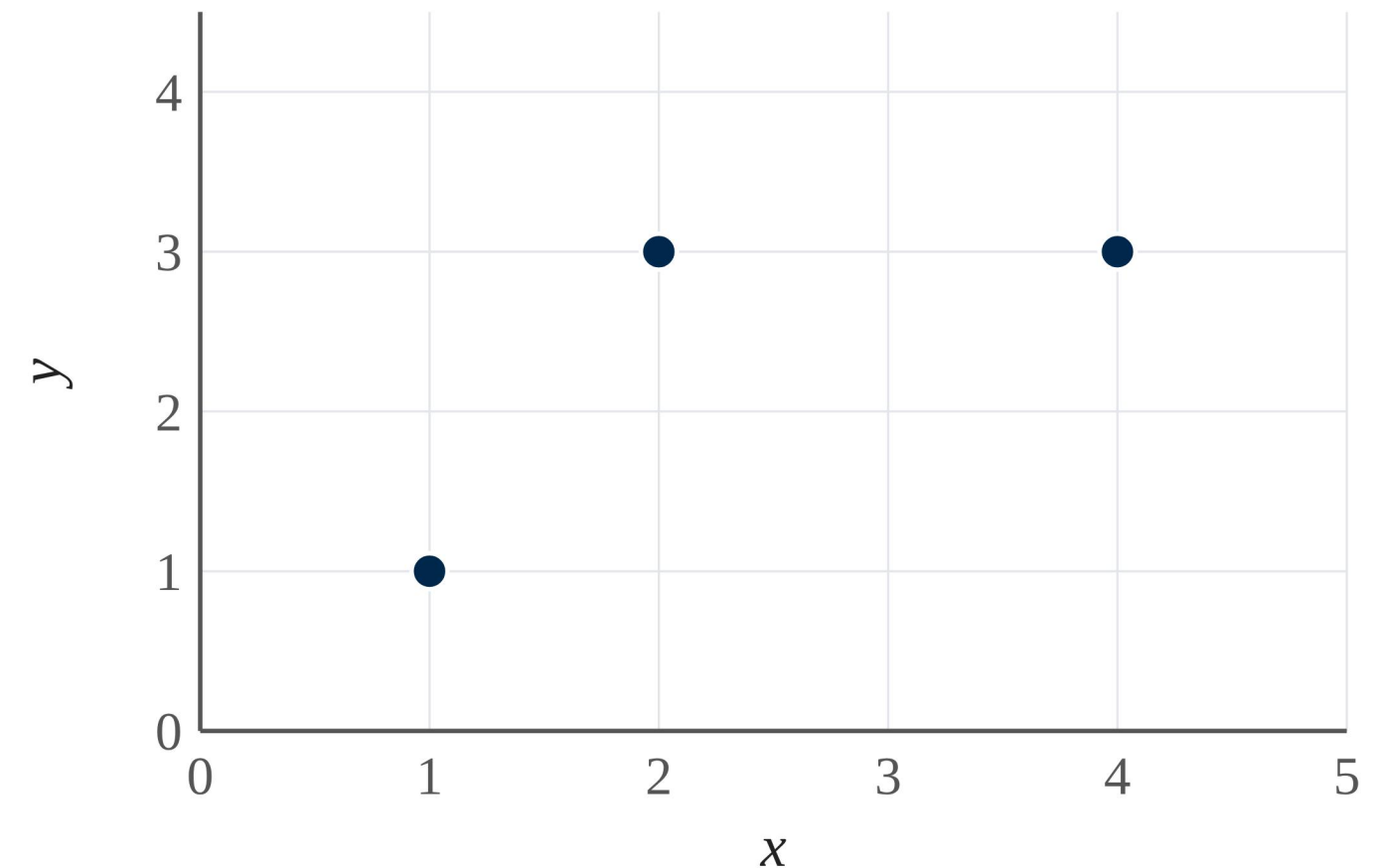
Let's start with a tiny regression problem. Our training data has three examples:

x	y
1	1
2	3
4	3

Each example is a pair (x, y) : an input, and its correct output.

We want a predictor f such that, e.g., $f(3)$ is a good guess for a new input $x = 3$.

Let's go through the three decisions.



Hypothesis class: which predictors?

Let's consider lines: $f(x) = w_1 + w_2x$.

For example, $f(x) = x$ ($w_1 = 0, w_2 = 1$)

or $f(x) = 1 + 0.57x$ ($w_1 = 1, w_2 = 0.57$).

In vector notation, put the **weights** in a vector
 $w = [w_1, w_2]$,

and define a **feature extractor** $\phi(x) = [1, x]$. Then:

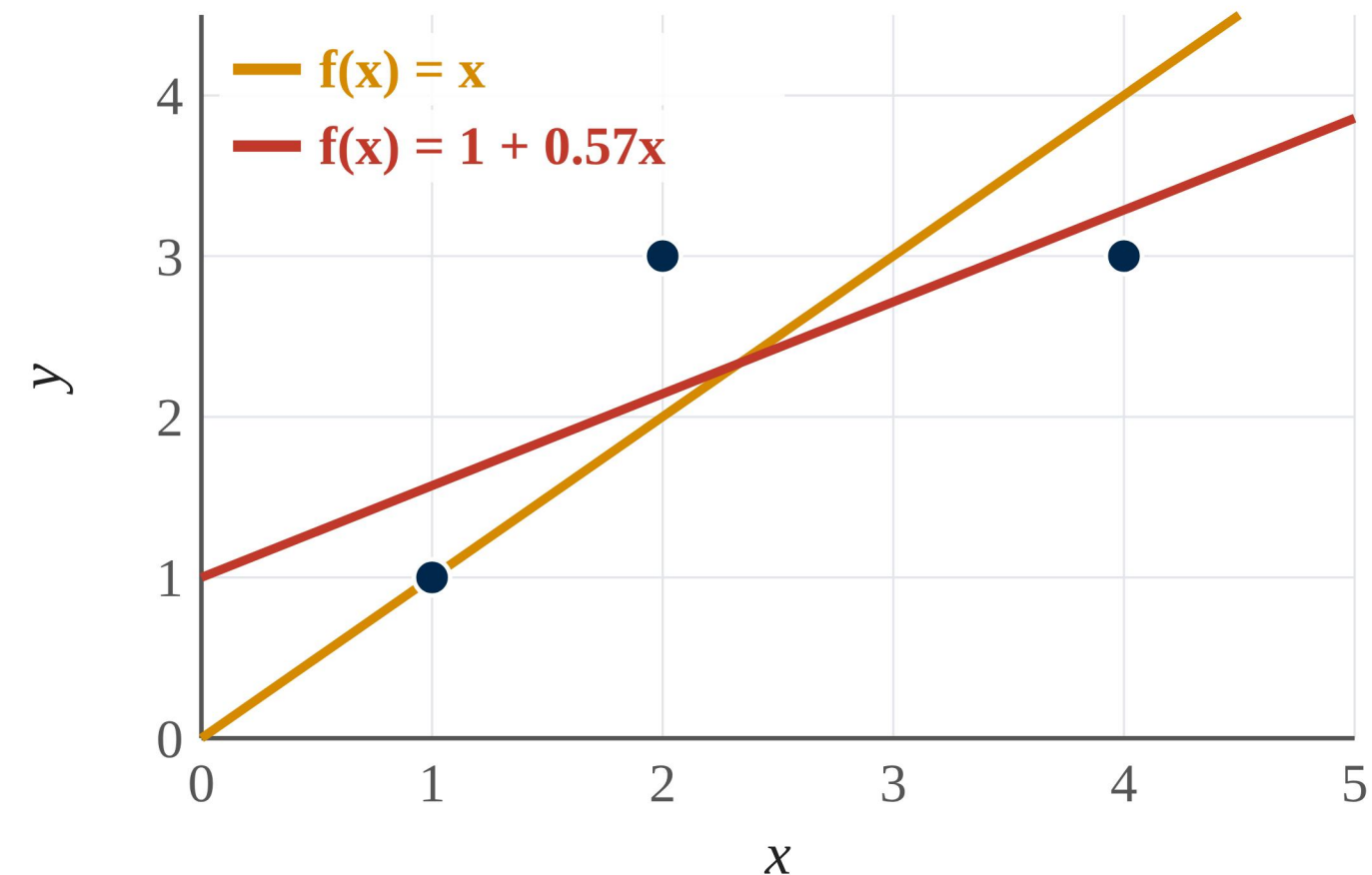
$$f_w(x) = w \cdot \phi(x)$$

We call $w \cdot \phi(x)$ the **score**. E.g.,

$$f_{[1, 0.57]}(3) = [1, 0.57] \cdot [1, 3] = 1 \cdot 1 + 0.57 \cdot 3 = 2.71.$$

The **hypothesis class** is the set of all predictors we'll consider:

$$\mathcal{F} = \{f_w : w \in \mathbb{R}^2\}$$



Loss function: how good is a predictor?

Intuitively, a good predictor fits the training data.

The **residual** $f_w(x) - y$ measures how far off a prediction is.

The **squared loss** on one example is:

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

The **training loss** averages it over the training set:

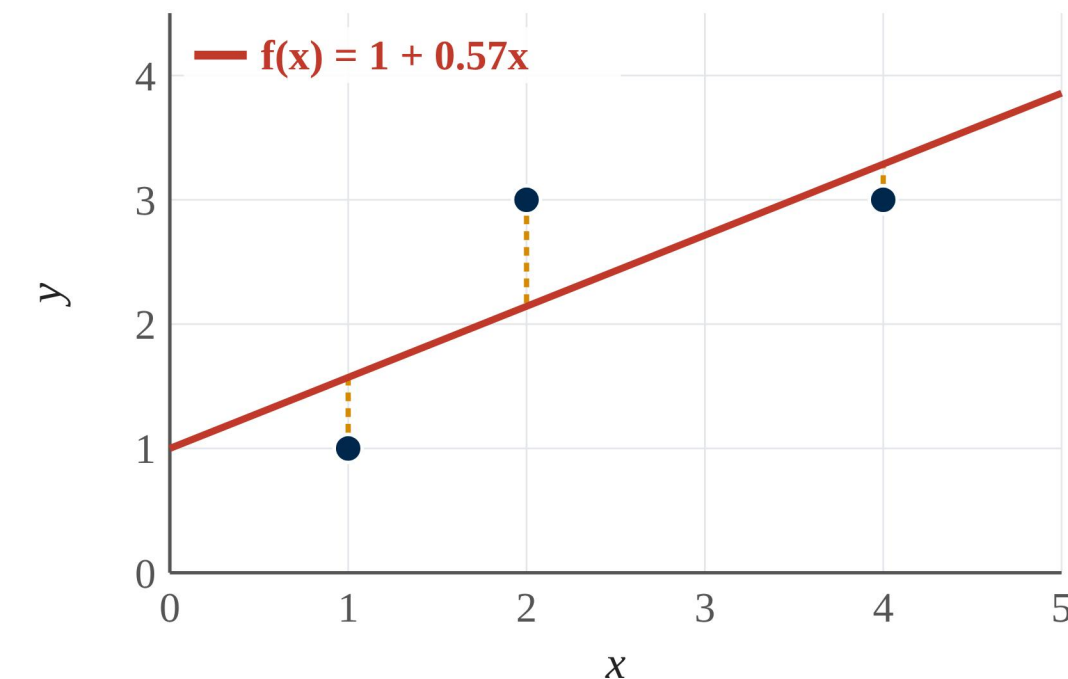
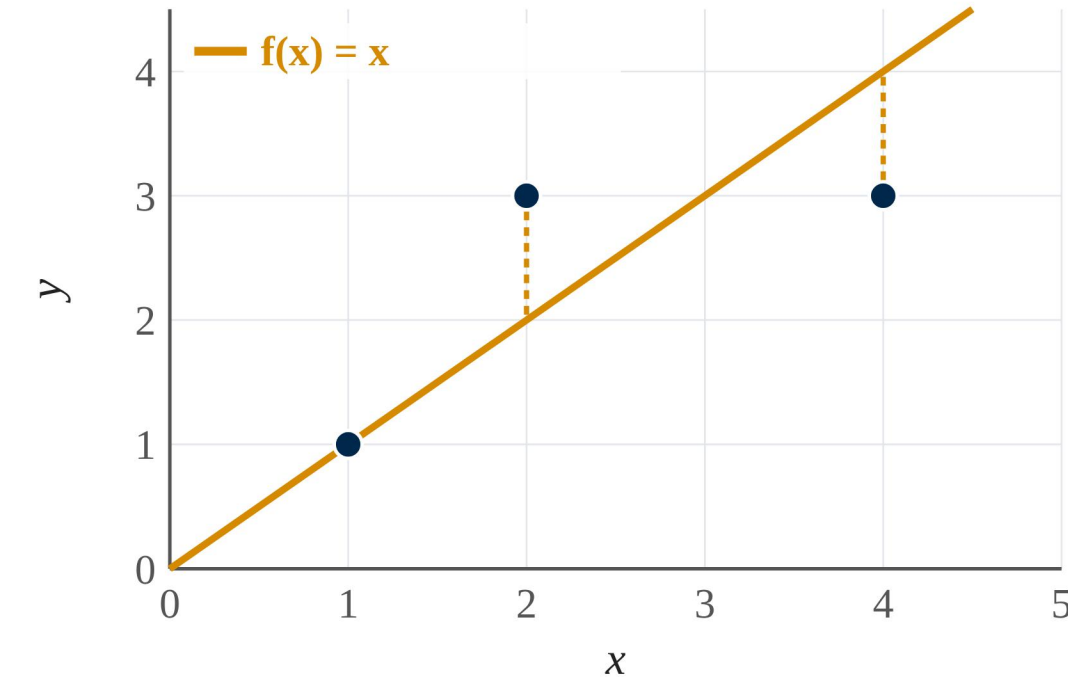
$$\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, w)$$

For $f(x) = x$, the residuals are 0, -1, 1:

$$\text{TrainLoss}([0, 1]) = \frac{0^2 + (-1)^2 + 1^2}{3} \approx .67$$

For $f(x) = 1 + 0.57x$, they are .57, -.86, .29:

$$\text{TrainLoss}([1, 0.57]) = \frac{.57^2 + .86^2 + .29^2}{3} \approx .38. \text{ Better!}$$



Where do squares come from?

Why square the residual, and not, e.g., absolute value?

Part of the answer is convenience: squares are smooth and easy to differentiate.

But there's also a probabilistic justification (again due to Gauss in 1809).

Suppose the true data is actually linear, but each observation has some random, independent Gaussian noise

$$y = w \cdot \phi(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma^2)$$

So, given x , the label we get y is Gaussian, centered at the line's prediction $w \cdot \phi(x)$

So it has density:

$$p(y \mid x; w) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left(-\frac{(y - w \cdot \phi(x))^2}{2\sigma^2} \right)$$

Which w makes our training data most likely? Like before, let's find the maximum likelihood estimate.

Since the examples are independent, the likelihood is a product:

$$L(w) = \prod_{i=1}^n p(y_i \mid x_i; w)$$

Least squares is maximum likelihood

Like before again, we can take logs to turn the product into a sum:

$$\begin{aligned}\log L(w) &= \sum_{i=1}^n \log p(y_i \mid x_i; w) \\ &= \sum_{i=1}^n \left(\log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(y_i - w \cdot \phi(x_i))^2}{2\sigma^2} \right) \\ &= -\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - w \cdot \phi(x_i))^2\end{aligned}$$

The first term doesn't depend on w . And $\frac{1}{2\sigma^2} > 0$, so maximizing $-\frac{1}{2\sigma^2}(\dots)$ means minimizing $(\dots)$:

$$\arg \max_w \log L(w) = \arg \min_w \sum_{i=1}^n (w \cdot \phi(x_i) - y_i)^2 = \arg \min_w \text{TrainLoss}(w)$$

So least squares is maximum likelihood, if we assume Gaussian noise. The noise level σ doesn't even matter!

Other noise models give other losses. With Laplace noise, $p(\varepsilon) \propto e^{-|\varepsilon|/b}$, the same steps give the absolute deviation $|f_w(x) - y|$, which is less sensitive to outliers.

The training loss landscape

Each weight vector $w = [w_1, w_2]$ has a training loss.

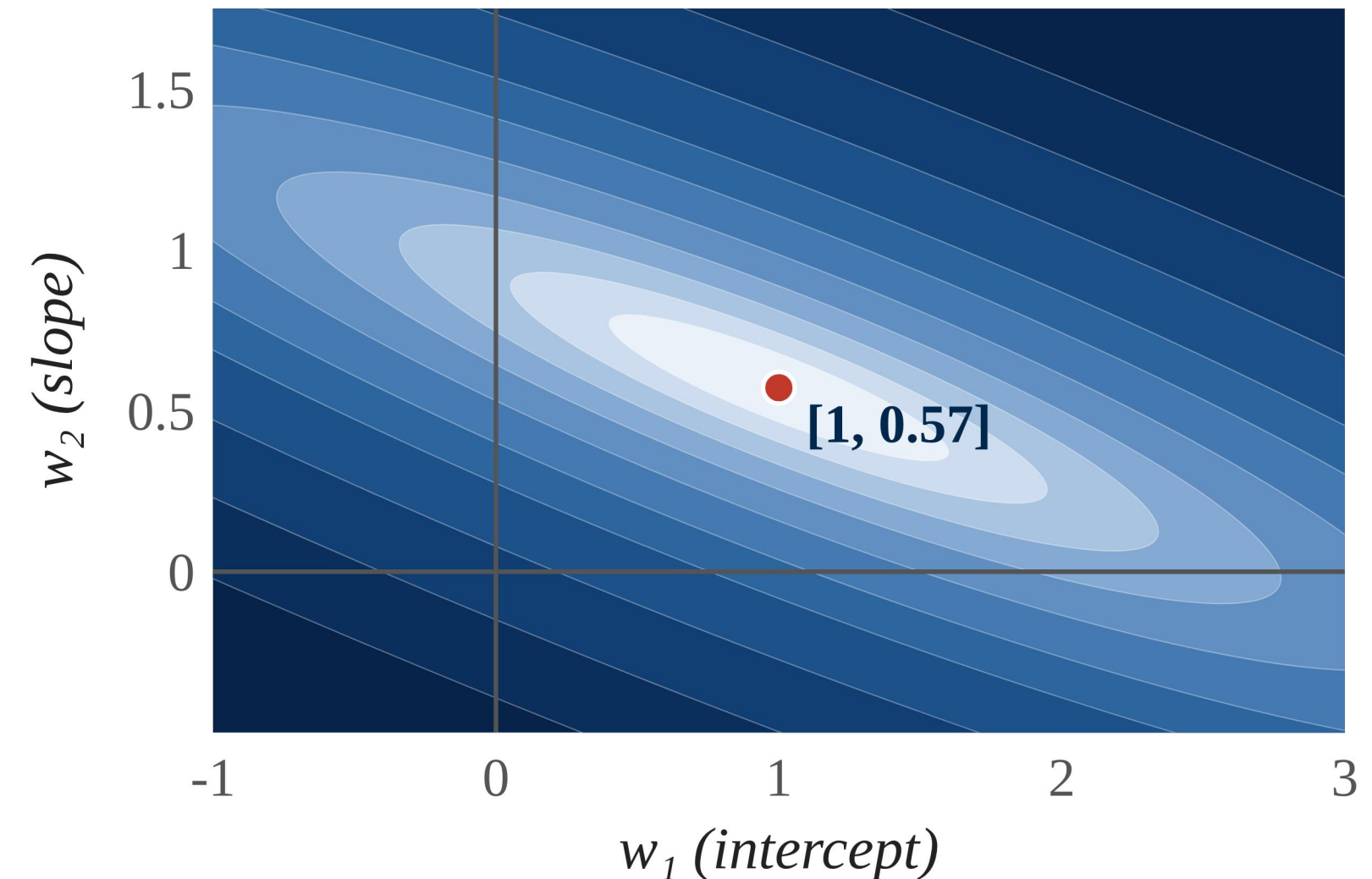
Since w is 2D here, we can plot it. (Darker = higher loss on same training set).

The best predictor in our class is the one with the lowest training loss:

$$\min_w \text{TrainLoss}(w)$$

Here, the minimum is at $w = [1, 0.57]$.

We now have a pure optimization problem. How do we solve it?



Optimization: gradient descent

Idea: start somewhere, and repeatedly take a small step downhill.

The gradient $\nabla_w \text{TrainLoss}(w)$ points in the direction of steepest ascent for the training loss.

So we move in the opposite direction:

```
1  function GRADIENT-DESCENT(TrainLoss,  $\eta$ , T):  
2       $w = [0, \dots, 0]$   
3      for  $t = 1, \dots, T$ :  
4           $w = w - \eta \nabla \text{TrainLoss}(w)$   
5      return  $w$ 
```

$\eta > 0$ is the **step size / learning rate**

T is the number of **epochs** (passes over the training data).

Both are **hyperparameters**: choices we make before learning, so not learned.

Computing the gradient

For linear regression with the squared loss:

$$\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (w \cdot \phi(x) - y)^2$$

Gradient of sum = sum of gradients. For each term, the chain rule gives:

$$\begin{aligned} \nabla_w (w \cdot \phi(x) - y)^2 &= 2(w \cdot \phi(x) - y) \nabla_w (w \cdot \phi(x) - y) \\ &= 2(w \cdot \phi(x) - y) \nabla_w (w \cdot \phi(x)) \\ &= 2(w \cdot \phi(x) - y) \phi(x) \nabla_w (w) \\ &= 2(w \cdot \phi(x) - y) \phi(x) \mathbb{I} \\ &= \underbrace{2(w \cdot \phi(x) - y) \phi(x)}_{\text{residual}} \end{aligned}$$

Averaging over the training set:

$$\nabla_w \text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2(w \cdot \phi(x) - y) \phi(x)$$

Each example pushes w in the direction of $\phi(x)$, in proportion to how wrong its prediction is.

An example we already predict exactly contributes 0.

Gradient descent on our example

We start with $w = [0, 0]$. Let's use $\eta = 0.1$.

The first gradient is:

$$\begin{aligned} & \frac{1}{3} (2(0 - 1)[1, 1] + 2(0 - 3)[1, 2] + 2(0 - 3)[1, 4]) \\ &= \frac{1}{3} [-14, -38] = [-4.67, -12.67] \end{aligned}$$

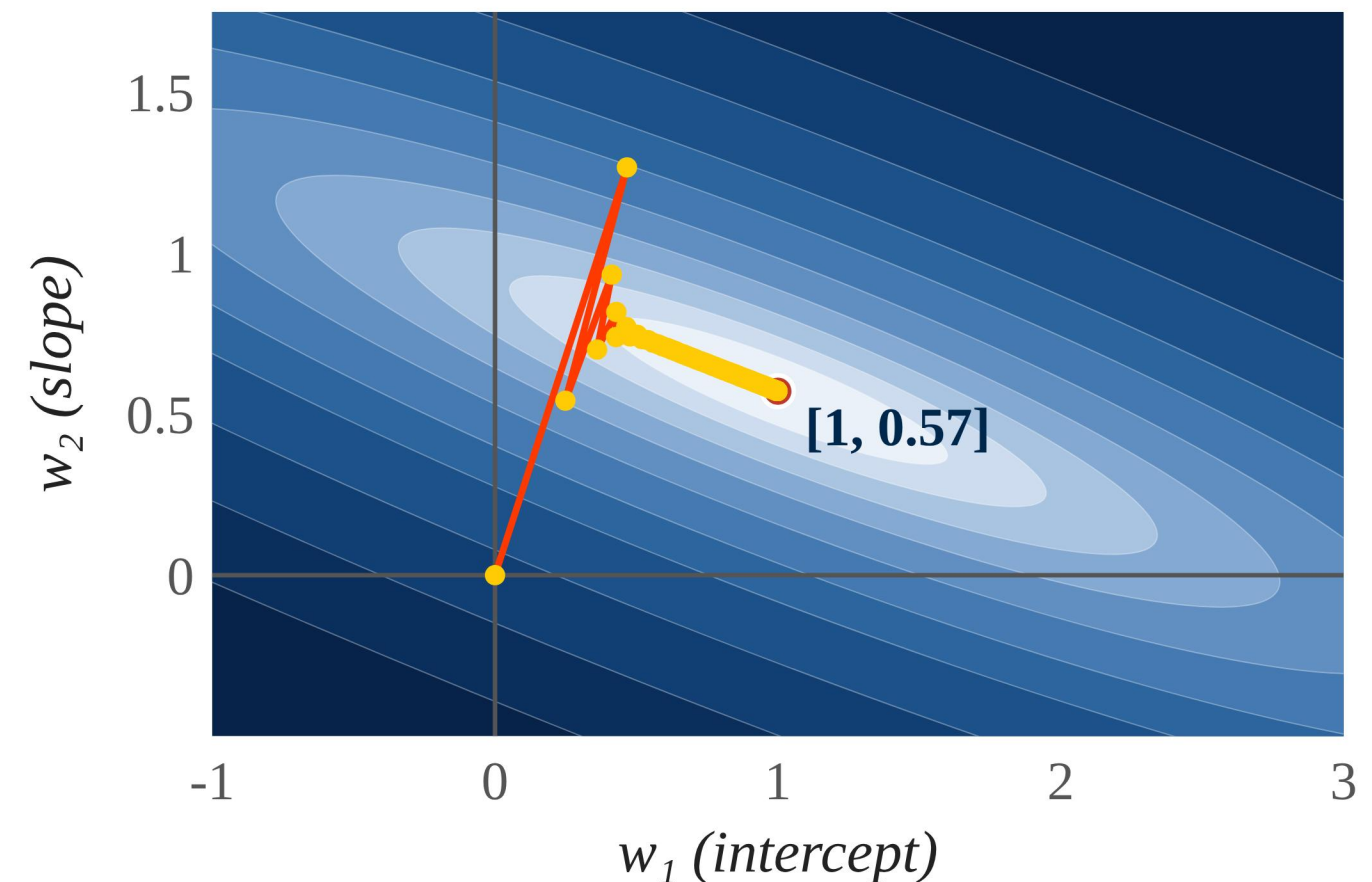
So $w \leftarrow [0, 0] - 0.1 \cdot [-4.67, -12.67] = [0.47, 1.27]$.

And again:

t	w	TrainLoss	Gradient
0	$[0, 0]$	6.33	$[-4.67, -12.67]$
1	$[0.47, 1.27]$	2.32	$[2.18, 7.24]$
2	$[0.25, 0.54]$	1.05	$[-1.64, -3.91]$
3	$[0.41, 0.93]$	0.65	$[0.52, 2.33]$
10	$[0.52, 0.73]$	0.43	$[-0.20, 0.03]$
60	$[0.94, 0.59]$	0.38	$[-0.03, 0.01]$

The slope overshoots back and forth at first, but the loss goes down at every step.

At the minimum $[1, 0.57]$ the gradient is zero, so gradient descent stops moving.



Separating the problem from the algorithm

Note that gradient descent never used the fact that we're fitting lines.

All it needs is a way to compute $\nabla_w \text{TrainLoss}(w)$.

If we change the features ϕ , or the loss, the same algorithm still works.

Least squares is special: it has a closed form (normal equations). Most losses we'll see don't.

Gradient-based optimization is what we'll use from here all the way to training neural networks and LLMs.

Stochastic gradient descent

Each step of gradient descent goes through the entire training set.

With a million examples, we need a million per-example gradients just to take one step!

But the training loss is an average. The gradient on a single example is a noisy estimate of the full gradient.

Stochastic gradient descent (SGD) updates w after each example:

```
1  function SGD(D_train, Loss,  $\eta$ , T):  
2       $w = [0, \dots, 0]$   
3      for  $t = 1, \dots, T$ :  
4          for  $(x, y)$  in D_train:    # in random order  
5               $w = w - \eta \nabla \text{Loss}(x, y, w)$   
6      return  $w$ 
```

Each update is noisier, but we make many more of them in the same amount of time.

In between: **minibatch SGD** averages the gradient over a batch of B examples in each update. This is how neural networks are trained in practice.

Let's implement gradient descent and SGD for linear regression, and try a few step sizes!

Choosing the step size

What should η be?

Too large: we overshoot, oscillate, and can even diverge.

Too small: stable, but slow.

On our example, $\eta = 0.1$ converges. With $\eta = 0.15$, after 60 steps the training loss is about 10^{16} !

Common choices: a constant ($\eta = 0.1$), or one that decreases over time, like $\eta_t = 1/\sqrt{t}$.

Fancier methods can adapt the step size dynamically, per parameter (e.g., Adam). (Soon in the Deep Learning module)

Now suppose $y \in \{+1, -1\}$ (binary classification).

We can keep the linear score $w \cdot \phi(x)$, and use its sign to predict:

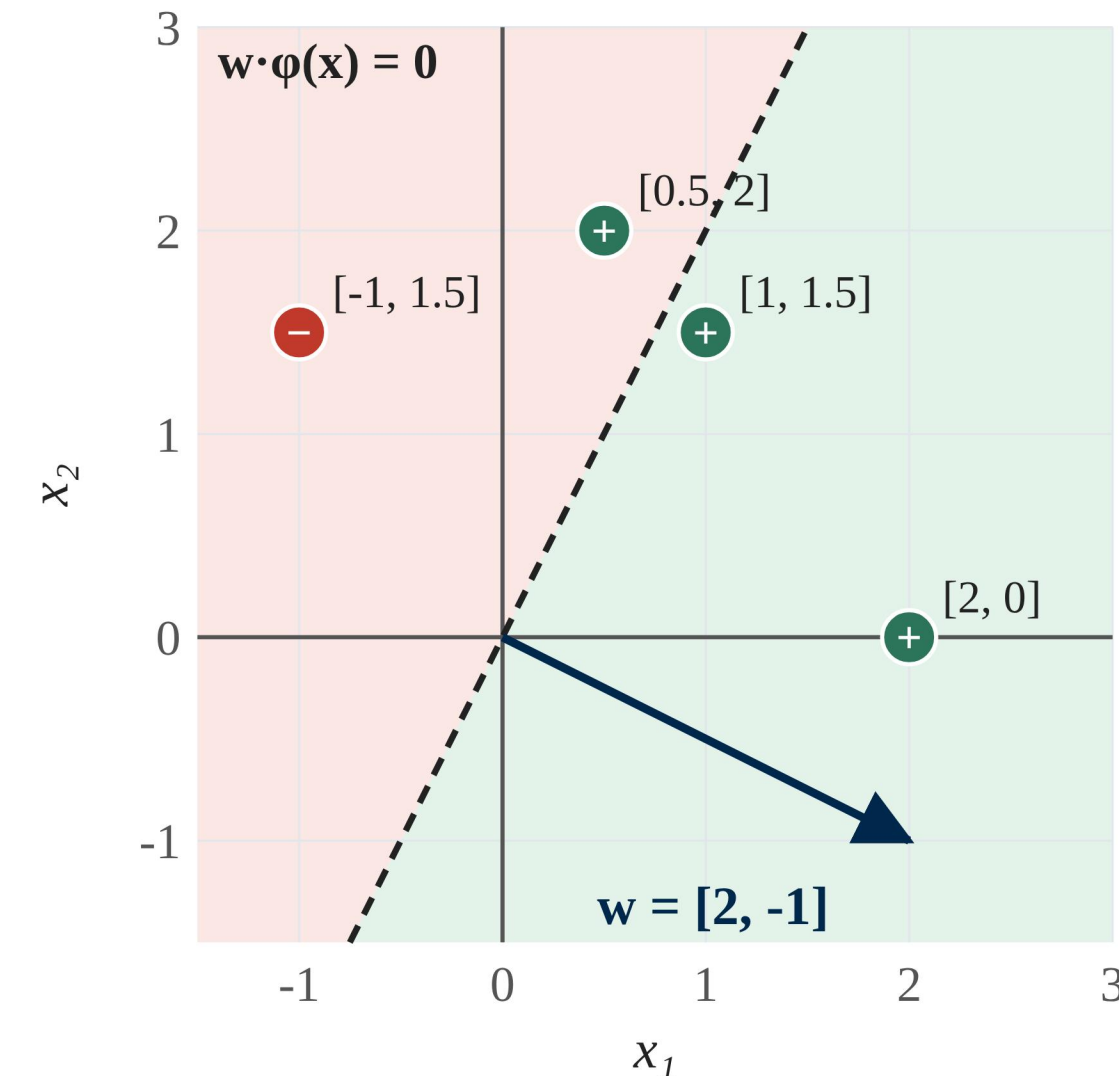
$$f_w(x) = \text{sign}(w \cdot \phi(x))$$

The inputs where the score is zero form the **decision boundary**. With $\phi(x) = [x_1, x_2]$, it's a line.

Which loss? Simplest one is to count mistakes:
 $\mathbf{1}[f_w(x) \neq y]$.

But bad for gradient descent: changing w a little almost never changes a prediction, so the gradient is zero almost everywhere!

We'll see how to fix this next lecture (and Homework 2)



GENERALIZATION

Is low training loss enough?

So far, our goal was to minimize the training loss.

Consider this learning algorithm:

Memorize the training set. Given x : if it's in $\mathcal{D}_{\text{train}}$, return its y . Otherwise, return 0.

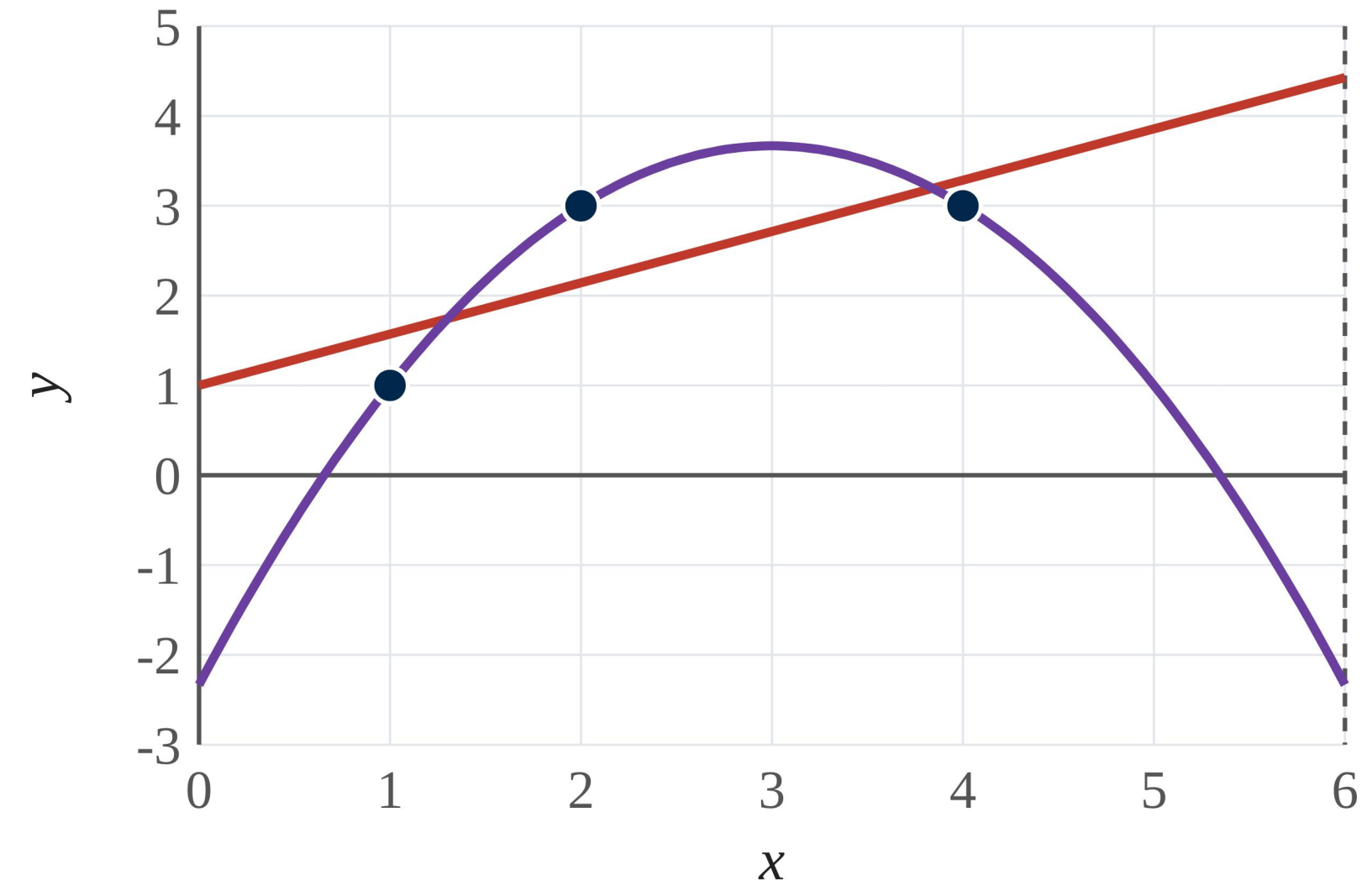
Zero training loss! But useless on any new input.

A less silly example: use $\phi(x) = [1, x, x^2]$ on our regression data.

The parabola $f(x) = -\frac{7}{3} + 4x - \frac{2}{3}x^2$ goes through all three points: zero training loss.

But it predicts $f(6) \approx -2.3$, while the line predicts 4.4. Which one would you trust?

This is **overfitting**: fitting the training data so closely that we do worse on new data.



What we actually want

We care about inputs we haven't seen yet. But we can't optimize for data we don't have!

Can we get any guarantees that we can generalize?

That needs some assumption connecting the training data to future data

Standard one: train + test data are i.i.d (independent and identically distributed)

Then what we want is a low **expected loss** (also called risk, or test error) on a new example:

$$\text{Err}(f) = \mathbb{E}_{(x,y) \sim \mathcal{D}} [\text{Loss}(x, y, f)]$$

We estimate it with a **test set**: examples held out from training, used only to evaluate the final predictor.

If we use the test set to make choices (features, hyperparameters, ...), it stops being a good estimate.

We can do that (e.g., hyperparameter tuning) using a separate **validation set**.

Generalization is the central problem in machine learning:

When does a low training loss imply a low test error?

- **Gibbs sampling:** use a Markov chain to approximate a joint by resampling one variable at a time

Then, learning to predict y from x directly:

- **Supervised learning:** learn a predictor f from examples (x, y) , to predict y for new inputs.
- **Three components:** hypothesis class, loss function, optimization method.
- **Linear predictors:** score $w \cdot \phi(x)$. Regression outputs the score, classification outputs the sign.
- **Squared loss:** least squares, which is maximum likelihood under Gaussian noise.
- **(Stochastic) gradient descent:** iterative optimization, only needs the gradient of the loss.
- **Generalization:** low training loss is not the goal. We want low error on new examples.

Next: when can we guarantee generalization, and moving beyond linear predictors (neural nets).