mlentary: basics of machine learning

These are optional notes for 6.86x. You can do all your assignments without these notes. These notes are here as a study aid. They are terse and don't cover all our topics. I'm happy to answer questions about the notes on Piazza. If you want to help improve these notes, ask me; I'll be happy to list your name among the contributors to these notes!

Clickable Table of Contents

A. prologue

what is learning?

By the end of this section, you'll be able to

- recognize whether a learning task fits the paradigm of *learning from examples* and whether it's *supervised* or *unsupervised*.
- identify within a completed learning-from-examples project: the *training inputs(outputs)*, *testing inputs(outputs)*, *hypothesis class*, *learned hypothesis*; and describe which parts depend on which.

KINDS OF LEARNING — How do we communicate patterns of desired behavior? We can teach:

by instruction: "to tell whether a mushroom is poisonous, first look at its gills..."

by example: "here are six poisonous fungi; here, six safe ones. see a pattern?"

by reinforcement: "eat foraged mushrooms for a month; learn from getting sick."

Machine learning is the art of programming computers to learn from such sources. We'll focus on the most important case: **learning from examples**.°

← Food For Thought: What's something you've learned by instruction? By example? By reinforcement? In Unit 5 we'll see that learning by example unlocks the other modes of learning.

FROM EXAMPLES TO PREDICTIONS — For us, a pattern of desired behavior is a function that for each given situation/prompt returns a favorable action/answer. We seek a program that, from a list of examples of prompts and matching answers, determines an underlying pattern. Our program is a success if this pattern accurately predicts answers for new, unseen prompts. We often define our program as a search, over some class $\mathcal{H}$ of candidate patterns (jargon: **hypotheses**), to maximize some notion of "intrinsic-plausibility plus goodness-of-fit-to-the-examples".

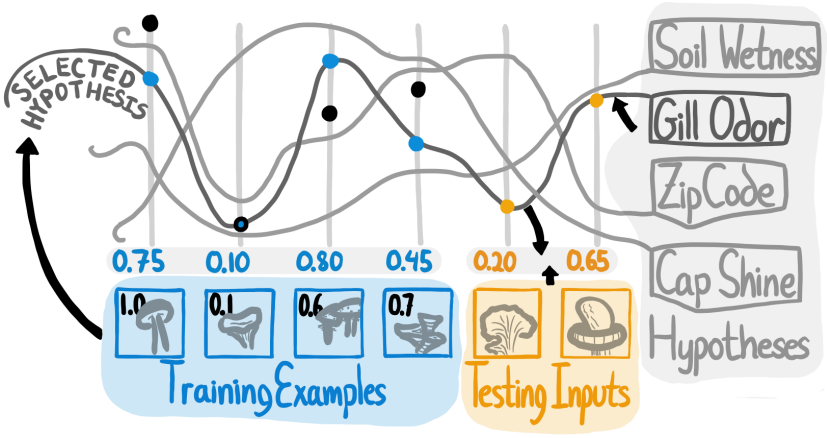


Figure 1: **Predicting mushrooms' poisons.** Our learning program selects from a class of hypotheses (gray blob) a plausible hypothesis that well fits (**blue dots** are close to **black dots**) a given list of poison-labeled mushrooms (blue blob). Evaluating the selected hypothesis on new mushrooms, we predict the corresponding poison levels (orange numbers).

The arrows show dataflow: how the hypothesis class and the mushroom+poisonlevel examples determine one hypothesis, which, together with new mushrooms, determines predicted poison levels. Selecting a hypothesis is called **learning**; predicting unseen poison levels, **inference**. The examples we learn from are **training data**; the new mushrooms and their true poison levels are **testing data**.

For example, say we want to predict poison levels (answers) of mushrooms (prompts). Among our hypotheses,° the GillOdor hypothesis fits the examples well: it guesses poison levels close to the truth. So the program selects GillOdor.

'Wait!', you say, *'doesn't Zipcode fit the example data more closely than GillOdor?'*. Yes. But a poison-zipcode proportionality is implausible: we'd need more evidence before believing Zipcode. We can easily make many oddball hypotheses; by chance some may fit our data well, but they probably won't predict well! Thus "intrinsic plausibility" and

← We choose four hypotheses: respectively, that a mushroom's poison level is close to:

— its ambient soil's percent water by weight;

— its gills' odor level, in kilo-Scoville units;

— its zipcode (divided by 100000);

— the fraction of visible light its cap reflects.

"goodness-of-fit-to-data" *both*° play a role in learning.

In practice we'll think of each hypothesis as mapping mushrooms to *distributions* over poison levels; then its "goodness-of-fit-to-data" is simply the chance it allots to the data.° We'll also use huge $\mathcal{H}$s: we'll *combine* mushroom features (wetness, odor, and shine) to make more hypotheses such as $(1.0 \cdot \text{GillOdor} - 0.2 \cdot \text{CapShine})$.° Since we can't compute "goodness-of-fit" for so many hypotheses, we'll guess a hypothesis then repeatedly nudge it up the "goodness-of-fit" *slope*.°

← We choose those two notions (and our $\mathcal{H}$) based on **domain knowledge**. This design process is an art; we'll study some rules of thumb.

← That's why we'll need **probability**.

← That's why we'll need **linear algebra**.

← That's why we'll need **derivatives**.

SUPERVISED LEARNING — We'll soon allow uncertainty by letting patterns map prompts to *distributions* over answers. Even if there is only one prompt — say, "*produce a beautiful melody*" — we may seek to learn the complicated distribution over answers, e.g. to generate a diversity of apt answers. Such **unsupervised learning** concerns output structure. By contrast, **supervised learning** (our main subject), concerns the input-output relation; it's interesting when there are many possible prompts. Both involve learning from examples; the distinction is no more firm than that between sandwiches and hotdogs, but the words are good to know.

our first learning algorithm

By the end of this section, you'll be able to

- write a simple, inefficient image classifier
- visualize data as lying in feature space; visualize hypotheses as functions defined on feature space; and visualize the class of all hypotheses within weight space

MEETING THE DATA — Say we want to classify handwritten digits. In symbols: we'll map $\mathcal{X}$ to $\mathcal{Y}$ with $\mathcal{X} = \{\text{grayscale } 28\times28\text{-pixel images}\}$, $\mathcal{Y} = \{1, 3\}$. Each datum (x, y) arises as follows: we randomly choose a digit $y \in \mathcal{Y}$, ask a human to write that digit in pen, and then photograph their writing to produce $x \in \mathcal{X}$.

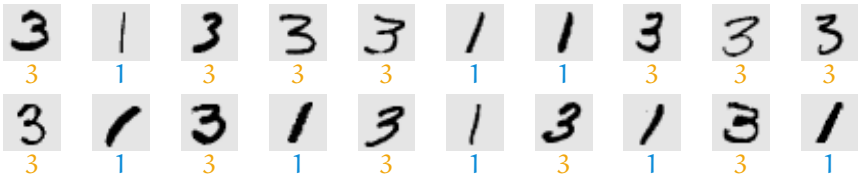


Figure 2: Twenty example pairs. Each photo x is a 28×28 grid of numbers representing pixel intensities. The light gray background has intensity 0.0; the blackest pixels, intensity 1.0. Below each photo x we display the corresponding label y: either $y = 1$ or $y = 3$. We'll adhere to this color code throughout this tiny

When we zoom in, we can see each photo's 28×28 grid of pixels. On the computer, this data is stored as a 28×28 grid of numbers: 0.0 for bright through 1.0 for dark. We'll name these 28×28 grid locations by their row number (counting from the top) followed by their column number (counting from the left). So location $(0, 0)$ is the upper left corner pixel; $(27, 0)$, the lower left corner pixel.

Food For Thought: Where is location $(0, 27)$? Which way is $(14, 14)$ off-center?

To get to know the data, let's wonder how we'd hand-code a classifier (worry not: soon we'll do this more automatically). We want to complete the code

```
def hand_coded_predict(x):
  return 3 if condition(x) else 1
```

Well, 3s tend to have more ink than than 1s — should `condition` threshold by the photo's brightness? Or: 1s and 3s tend to have different widths — should `condition` threshold by the photo's dark part's width?

To make this precise, let's define a photo's *brightness* as 1.0 minus its average pixel darkness; its *width* as the standard deviation of the column index of its dark pixels. Such functions from inputs in $\mathcal{X}$ to numbers are called **features**.

```
SIDE = 28
def brightness(x):  return 1. - np.mean(x)
def width(x):       return np.std([col for col in range(SIDE)
                                       for row in range(SIDE)
                                       if 0.5 < x[row][col]  ])/(SIDE/2.0)
# (we normalized width by SIDE/2.0 so that it lies within [0., 1.])
```

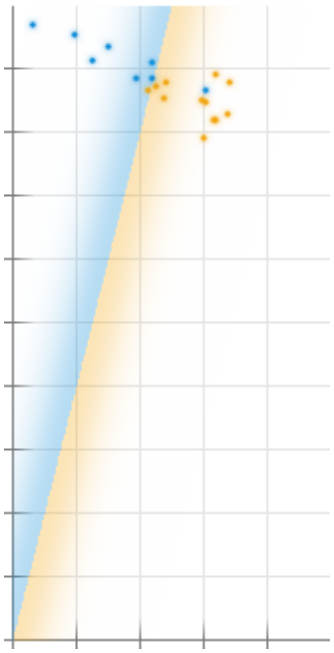


Figure 3: **Featurized training data.** Our $N = 20$ many training examples, viewed in the brightness-width plane. The vertical *brightness* axis ranges $[0.0, 1.0]$; the horizontal *width* axis ranges $[0.0, 0.5]$. The origin is at the lower left. Orange dots represent $y = 3$ examples; blue dot, $y = 1$ examples. We eyeballed the line $-1 \cdot \text{brightness} + 4 \cdot \text{width} = 0$ to separate the two kinds of examples.

So we can threshold by brightness or by width. But this isn't very satisfying, since sometimes there are especially dark 1s or thin 3s. Aha! Let's use *both* features: 3s are darker than 1s *even relative to their width*. Inspecting the training data, we see that a line through the origin of slope 4 roughly separates the two classes. So let's threshold by a combination like `-1*brightness(x)+4*width(x)`:

```
def condition(x):
  return -1*brightness(x)+4*width(x) > 0
```

Intuitively, the formula $-1 \cdot \text{brightness} + 4 \cdot \text{width}$ we invented is a measure of *threeness*: if it's positive, we predict $y = 3$. Otherwise, we predict $y = 1$.

Food For Thought: What further features might help us separate digits 1 from 3?

CANDIDATE PATTERNS — We can generalize the hand-coded hypothesis from the previous passage to other coefficients besides $-1 \cdot \text{brightness}(x) + 4 \cdot \text{width}(x)$. We let our set $\mathcal{H}$ of candidate patterns contain all "linear hypotheses" $f_{a,b}$ defined by:

$$f_{a,b}(x) = 3 \text{ if } a \cdot \text{brightness}(x) + b \cdot \text{width}(x) > 0 \text{ else } 1$$

Each $f_{a,b}$ makes predictions of ys given xs. As we change a and b, we get different predictors, some more accurate than others.

```
def predict(x,a,b):
  return 3 if a*brightness(x) + b*width(x) > 0 else 1
```

The brightness-width plane is called **feature space**: its points represent inputs x in terms of chosen features (here, brightness and width). The (a, b) plane is called **weight space**: its points represent linear hypotheses h in terms of the coefficients — or **weights** — h places on each feature (e.g. $a = -1$ on brightness and $b = +4$ on width).

Food For Thought: Which of Fig. 4's 3 hypotheses best predicts training data?

Food For Thought: What (a, b) pairs might have produced Fig. 4's 3 hypotheses? Can you determine (a, b) for sure, or is there ambiguity (i.e., can multiple (a, b) pairs make exactly the same predictions in brightness-width space)?

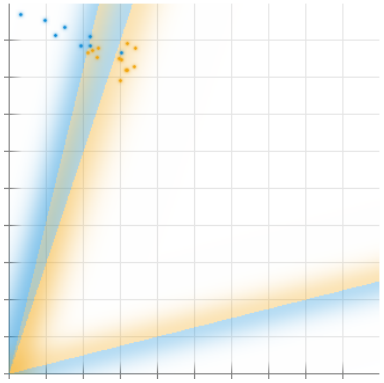


Figure 4: **Hypotheses differ in training accuracy: feature space.** 3 hypotheses classify training data in the brightness-width plane (axes range $[0, 1.0]$). Glowing colors distinguish a hypothesis' 1 and 3 sides. For instance, the bottommost line classifies all the training points as 3s. **Caution**: the colors in this page's two Figures represent unrelated distinctions!

OPTIMIZATION — Let's write a program to automatically find hypothesis $h = (a, b)$ from the training data. We want to predict the labels y of yet-unseen photos x (*testing examples*); insofar as training data is representative of testing data, it's sensible to return a $h \in \mathcal{H}$ that correctly classifies maximally many training examples.

To do this, let's just loop over a bunch (a, b)s — say, all integer pairs in $[-99, +99]$ — and pick one that misclassifies the least training examples:

```
def is_correct(x,y,a,b):
  return 1.0 if predict(x,a,b)==y else 0.0
def accuracy_on(examples,a,b):
  return np.mean(is_correct(x,y,a,b) for x,y in examples)
def best_hypothesis():
  # returns a pair (accuracy, hypothesis)
  return max((accuracy_on(training_data, a, b), (a,b))
             for a in np.arange(-99,+100)
             for b in np.arange(-99,+100)            )
```

Fed our $N = 20$ training examples, the loop finds $(a, b) = (-20, +83)$ as a minimizer of **training error**, i.e., of the fraction of training examples misclassified. It misclassifies only 10% of training examples. Yet the same hypothesis misclassifies a greater fraction — 17% — of fresh, yet-unseen testing examples. That latter number — called the **testing error** — represents our program's accuracy "in the wild"; it's the number we most care about.

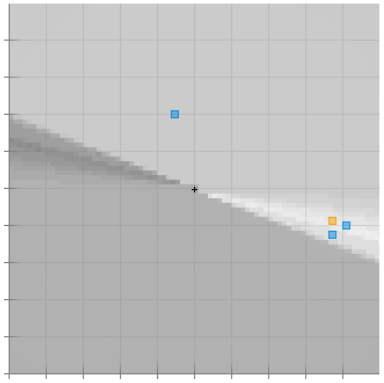


Figure 5: **Hypotheses differ in training accuracy: weight space.** We visualize $\mathcal{H}$ as the (a, b)-plane (axes range $[-99, +99]$). Each point determines a whole line in the brightness-width plane. Shading shows training error: darker points misclassify more training examples. The least shaded, most training-accurate hypothesis is $(-20, 83)$: the rightmost of the 3 blue squares. The orange square is the hypothesis that best fits our unseen testing data.

Food For Thought: Suppose Fig. 4's 3 hypotheses arose from Fig. 5's 3 blue squares. Which hypothesis matches each square?

The difference between training and testing error is the difference between our score on our second try on a practice exam (after we've reviewed our mistakes) versus our score on a real exam (where we don't know the questions beforehand and aren't allowed to change our answers once we get our grades back).

Food For Thought: In the (a, b) plane shaded by training error, we see two 'cones', one dark and one light. They lie geometrically opposite to each other — why?

Food For Thought: Sketch $f_{a,b}$'s error on $N = 1$ example as a function of (a, b).

how well did we do? (3 kinds of error : opt, approx, gen)

By the end of this section, you'll be able to

- compute and conceptually distinguish training and testing misclassification errors
- explain how the problem of achieving low testing error decomposes into the three problems of achieving low *generalization*, *optimization*, and *approximation* errors.

ERROR ANALYSIS — Intuitively, our testing error of 17% comes from three sources: **(a)** the failure of our training set to be representative of our testing set; **(b)** the failure of our program to exactly minimize training error over $\mathcal{H}$; and **(c)** the failure of our hypothesis set $\mathcal{H}$ to contain "the true" pattern.

These are respectively errors of **generalization**, **optimization**, **approximation**.

We can see generalization error when we plot testing data in the brightness-width plane. The hypotheses $h = (20, 83)$ that we selected based on the training in the brightness-width plane misclassifies many testing points. we see many misclassified points. Whereas h misclassifies only 10% of the training data, it misclassifies 17% of the testing data. This illustrates generalization error.

In our plot of the (a, b) plane, the blue square is the hypothesis h (in $\mathcal{H}$) that best fits the training data. The orange square is the hypothesis (in $\mathcal{H}$) that best fits the testing data. But even the latter seems suboptimal, since $\mathcal{H}$ only includes lines through the origin while it seems we want a line — or curve — that hits higher up on the brightness axis. This illustrates approximation error.°

← To define *approximation error*, we need to specify whether the 'truth' we want to approximate is the training or the testing data. Either way we get a useful concept. In this paragraph we're talking about approximating *testing* data; but in our notes overall we'll focus on the concept of error in approximating *training* data.

Optimization error is best seen by plotting training rather than testing data. It measures the failure of our selected hypothesis h to minimize training error — i.e., the failure of the blue square to lie in a least shaded point in the (a, b) plane, when we shade according to training error.

Here, we got optimization error $\approx 0\%$ (albeit by *unscalable brute-force*). Because optimization error is zero in our case, the approximation error and training error are the same: $\approx 10\%$. The approximation error is so high because our straight lines are *too simple*: brightness and width lose useful information and the "true" boundary between digits — even training — may be curved. Finally, our testing error $\approx 17\%$ exceeds our training error. We thus suffer a generalization error of $\approx 7\%$: we *didn't perfectly extrapolate* from training to testing situations.

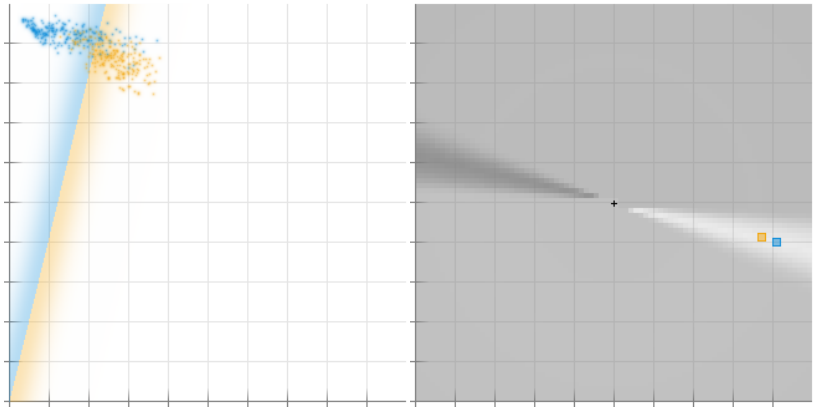


Figure 6: **Testing error visualized two ways.**. — **Left: in feature space.** The hypotheses $h = (20, 83)$ that we selected based on the training set classifies testing data in the brightness-width plane; glowing colors distinguish a hypothesis' 1 and 3 sides. Axes range $[0, 1.0]$. — **Right: in weight space.** Each point in the (a, b) plane represents a hypothesis; darker regions misclassify a greater fraction of testing data. Axes range $[-99, +99]$.

In 6.86x we'll address all three italicized issues.

Food For Thought: why is generalization error usually positive?

FORMALISM — Here's how we can describe learning and our error decomposition in symbols. Draw training examples $\mathcal{S} : (\mathcal{X} \times \mathcal{Y})^N$ from nature's distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$. A hypothesis $f : \mathcal{X} \to \mathcal{Y}$ has **training error** $\text{trn}_{\mathcal{S}}(f) = \mathbb{P}_{(x,y)\sim\mathcal{S}}[f(x) \neq y]$, an average over examples; and **testing error** $\text{tst}(f) = \mathbb{P}_{(x,y)\sim\mathcal{D}}[f(x) \neq y]$, an average over nature. A *learning program* is a function $\mathcal{L} : (\mathcal{X} \times \mathcal{Y})^N \to (\mathcal{X} \to \mathcal{Y})$; we want to design $\mathcal{L}$ so that it maps typical $\mathcal{S}$s to fs with low $\text{tst}(f)$.

VERY OPTIONAL

So we often define $\mathcal{L}$ to roughly minimize $\text{trn}_{\mathcal{S}}$ over a set $\mathcal{H} \subseteq (\mathcal{X} \to \mathcal{Y})$ of candidate patterns. Then tst decomposes into the failures of $\text{trn}_{\mathcal{S}}$ to estimate tst (generalization), of $\mathcal{L}$ to minimize $\text{trn}_{\mathcal{S}}$ (optimization), and of $\mathcal{H}$ to contain nature's truth (approximation):

$$\begin{aligned} \text{tst}(\mathcal{L}(\mathcal{S})) = {} & \text{tst}(\mathcal{L}(\mathcal{S})) && - \text{trn}_{\mathcal{S}}(\mathcal{L}(\mathcal{S})) && \} \textbf{ generalization} \text{ error} \\ & + \text{trn}_{\mathcal{S}}(\mathcal{L}(\mathcal{S})) && - \inf_{\mathcal{H}}(\text{trn}_{\mathcal{S}}(f)) && \} \textbf{ optimization} \text{ error} \\ & + \inf_{\mathcal{H}}(\text{trn}_{\mathcal{S}}(f)) && && \} \textbf{ approximation} \text{ error} \end{aligned}$$

These terms are in tension. For example, as $\mathcal{H}$ grows, the approx. error may decrease while the gen. error may increase — this is the "**bias-variance** tradeoff".

how can we do better? (survey of rest of notes)

B. auto-predict by fitting lines to examples (unit 1)

linear approximation

By the end of this section, you'll be able to

- define a class of linear, probabilistic hypotheses appropriate to a given classification task, by: designing features; packaging the coefficients to be learned as a matrix; and selecting a probability model (logistic, perceptron, SVM, etc).
- compute the loss suffered by a probabilistic hypothesis on given data

TWO THIRDS BETWEEN DOG AND COW — Remember: our Unit 1 motto is to *learn linearities flanked by hand-coded nonlinearities*:

$$\mathcal{X} \xrightarrow[\text{not learned}]{\text{featurize}} \mathbb{R}^2 \xrightarrow[\textbf{learned!}]{\text{linearly combine}} \mathbb{R}^1 \xrightarrow[\text{not learned}]{\text{read out}} \mathcal{Y}$$

We design the nonlinearities to capture domain knowledge about our data and goals. Here we'll design nonlinearities to help model *uncertainty* over $\mathcal{Y}$. We can do this by choosing a different read-out function. For example, representing distributions by objects `{3:prob_of_three, 1:prob_of_one}`, we could choose:

```
prediction = {3 : 0.8 if threeness[0]>0. else 0.2,
              1 : 0.2 if threeness[0]>0. else 0.8 }
```

If before we'd have predicted "the label is 3", we now predict "the label is 3 with 80% chance and 1 with 20% chance". This hard-coded 80% *could* suffice.° But let's do better: intuitively, a 3 is more likely when `threeness` is huge than when `threeness` is nearly zero. So let's replace that 80% by some smooth function of `threeness`. A popular, theoretically warranted choice is $\sigma(z) = 1/(1+\exp(-z))$:°

```
sigma = lambda z : 1./(1.+np.exp(-z))
prediction = {3 :    sigma(threeness[0]),
              1 : 1.-sigma(threeness[0]) }
```

Given training inputs x_i, a hypothesis will have "hunches" about the training outputs y_i. Three hypotheses $h_{\text{three!}}$, h_{three}, and h_{one} might, respectively, confidently assert $y_{42} = 3$; merely lean toward $y_{42} = 3$; and think $y_{42} = 1$. If in reality $y_{42} = 1$ then we'd say h_{one} did a good job, h_{three} a bad job, and $h_{\text{three!}}$ a very bad job on the 42nd example. So the training set "surprises" different hypotheses to different degrees. We may seek a hypothesis $h_\star$ that is minimally surprised, i.e., usually confidently right and when wrong not confidently so. In short, by outputting probabilities instead of mere labels, we've earned this awesome upshot: *the machine can automatically calibrate its confidence levels!* It's easy to imagine how important this calibration is in language, self-driving, etc.

Confidence on mnist example! (2 pictures, left and right: hypotheses and (a,b plane)

INTERPRETING WEIGHTS — We note two aspects of the 'intuitive logic' of weights.

Just because two features both correlate with a positive label ($y = +1$) doesn't mean both features will have positive weights. In other words, it could be that the *blah*-feature correlates with $y = +1$ in the training set and yet, according to the best hypothesis for that training set, the bigger a fresh input's blah feature is, the *less* likely its label is to be $+1$, all else being equal. That last phrase "all else being equal" is crucial, since it refers to our choice of coordinates. See the figure, left three panels.

Note on interpreting weights

Moreover, transforming coordinates, even linearly, can alter predictions. For example, if we shear two features together — say, by using cooktime-plus-preptime and cooktime as features rather than preptime and cooktime as features — this can impact the decision boundary.

← As always, it depends on what specific thing we're trying to do!

← σ, the **logistic** or **sigmoid** function, has linear log-odds: $\sigma(z)/(1-\sigma(z)) = \exp(z)/1$. It tends exponentially to the step function. It's symmetrical: $\sigma(-z) = 1-\sigma(z)$. Its derivative concentrates near zero: $\sigma'(z) = \sigma(z)\sigma(-z)$.
Food For Thought: Plot $\sigma(z)$ by hand.

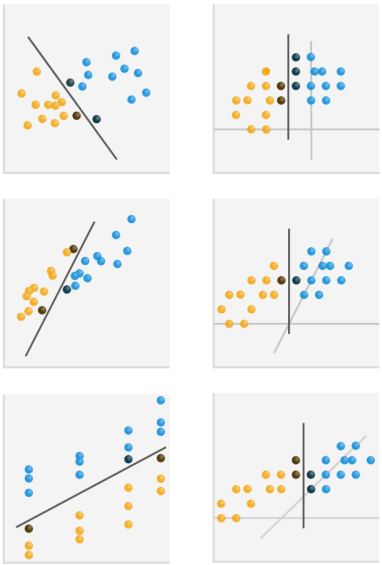


Figure 7: **Relations between feature statistics and optimal weights.** Each of these six figures shows a different binary classification task along with a maximum-margin hypothesis. We shade the datapoints that achieve the margin. — **Left:** *positive weights don't imply positive correlation!* — **Right:** *presenting the same information in different coordinates alters predictions!*

Of course, the decision boundary will look different because we're in new coordinates; but we mean something more profound: if we train in old coordinates and then predict a datapoint represented in old coordinates, we might get a different prediction than if we train in new coordinates and then predict a datapoint represented in new coordinates! See the figure, right three panels: here, the intersection of the two gray lines implicitly marks a testing datapoint that experiences such a change of prediction as we adopt different coordinates. *Intuitively, the more stretched out a feature axis is, the more the learned hypothesis will rely on that feature.*

Food For Thought: Understand this paragraph from the point of view of the L2 regularizer.

DESIGNING FEATURIZATIONS — We represent our input x as a fixed-length list of numbers so that we can "do math" to x. For instance, we could represent a 28×28 photo by 2 numbers: its overall brightness and its dark part's width. Or we could represent it by 784 numbers, one for the brightness at each of the $28 \cdot 28 = 784$ many pixels. Or by 10 numbers that respectively measure the overlap of x's ink with that of "representative" photos of the digits 0 through 9.

A way to represent x as a fixed-length list of numbers is a **featurization**. Each map from raw inputs to numbers is a **feature**. Different featurizations make different patterns easier to learn. We judge a featurization not in a vacuum but with respect to the kinds of patterns we use it to learn. information easy for the machine to use (e.g. through apt nonlinearities) and throw away task-irrelevant information (e.g. by turning 784 pixel brightnesses to 2 meaningful numbers).

Here are two themes in the engineering art of featurization.[◦]

[◦] For now, we imagine hand-coding our features rather than adapting them to training data. We'll later discuss adapted features; simple examples include thresholding into **quantiles** based on sorted training data (*Is x more than the median training point?*), and choosing coordinate transforms that measure similarity to **landmarks** (*How far is x from each of these 5 "representative" training points?*). Deep learning is a fancy example.

Predicates. If domain knowledge suggests some subset $S \subseteq \mathcal{X}$ is salient, then we can define the feature

$$x \mapsto 1 \text{ if } x \text{ lies in } S \text{ else } 0$$

The most important case helps us featurize *categorical* attributes (e.g. kind-of-chess-piece, biological sex, or letter-of-the-alphabet): if an attribute takes K possible values, then each value induces a subset of $\mathcal{X}$ and thus a feature. These features assemble into a map $\mathcal{X} \to \mathbb{R}^K$. This **one-hot encoding** is simple, powerful, and common. Likewise, if some attribute is *ordered* (e.g. $\mathcal{X}$ contains geological strata) then interesting predicates may include **thresholds**.

Coordinate transforms. Applying our favorite highschool math functions gives new features $\tanh(x[0]) - x[1]$, $|x[1]x[0]| \exp(-x[2]^2)$, $\cdots$ from old features $x[0], x[1], \cdots$. We choose these functions based on domain knowledge; e.g. if $x[0], x[1]$ represent two spatial positions, then the distance $|x[0] - x[1]|$ may be a useful feature. One systematic way to include nonlinearities is to include all the monomials (such

as $x[0]x[1]^2$) with not too many factors — then linear combinations are polynomials The most important nonlinear coordinate transform uses all monomial features with 0 or 1 many factors — said plainly, this maps

$$x \mapsto (1, x)$$

This is the **bias trick**. Intuitively, it allows the machine to learn the threshold above which three-ishness implies a three.

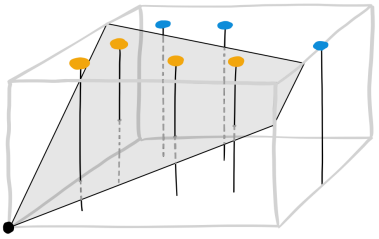


Figure 8: **The bias trick helps us model 'offset' decision boundaries.** Here, the origin is the lower right corner closer to the camera. Our raw inputs $x = (x[0], x[1])$ are 2-dimensional; we can imagine them sitting on the bottom face of the plot (bottom ends of the vertical stems). But, within that face, no line through the origin separates the data well. By contrast, when we use a featurization $(1, x[0], x[1])$, our data lies on the top face of the plot; now a plane through the origin (shown) successfully separates the data.

curvy trick

HUMBLE MODELS — Let's modify logistic classification to allow for *unknown unknowns*. We'll do this by allowing a classifier to allot probability mass not only among labels in $\mathcal{Y}$ but also to a special class $\star$ that means "no comment" or "alien input". A logistic classifier always sets $p_{y|x}[\star|x] = 0$, but other probability models may put nonzero mass on "no comment". For example, consider:

	LOGISTIC	PERCEPTRON	SVM
$p_{y\|x}[+1\|x]$	$\oplus/(\ominus+\oplus)$	$\oplus\cdot(\ominus\wedge\oplus)/2$	$\oplus\cdot(\ominus\wedge\oplus/e)/2$
$p_{y\|x}[-1\|x]$	$\ominus/(\ominus+\oplus)$	$\ominus\cdot(\ominus\wedge\oplus)/2$	$\ominus\cdot(\ominus/e\wedge\oplus)/2$
$p_{y\|x}[\star\|x]$	$1-$ above $=0$	$1-$ above	$1-$ above
outliers	responsive	robust	robust
inliers	sensitive	blind	sensitive
acc bnd	good	bad	good
loss name	softplus$(\cdot)$	srelu$(\cdot)$	hinge$(\cdot)$
formula	$\log_2(1+e^{(\cdot)})$	$\max(1,\cdot)+1$	$\max(1,\cdot+1)$
update	$1/(1+e^{+y\mathfrak{d}})$	step$(-y\mathfrak{d})$	step$(1-y\mathfrak{d})$

Table 1: **Three popular models for binary classification. Top rows:** Modeled chance given x that $y = +1, -1, \star$. We use $\mathfrak{d} = \vec{w}\cdot\vec{x}$, $\oplus = e^{+\mathfrak{d}/2}, \ominus = e^{-\mathfrak{d}/2}$, $a\wedge b = \min(a,b)$ to save ink. **Middle rows:** All models respond to misclassifications. But are they robust to well-classified outliers? Sensitive to well-classified inliers? **Bottom rows:** For optimization, which we'll discuss later, we list (negative log-probability) losses. An SGD step looks like

$$\vec{w}_{t+1} = \vec{w}_t + \eta\cdot\text{update}\cdot y\vec{x}$$

graphs of prob

MLE with the perceptron model or svm model minimizes the same thing, but with $\text{srelu}(z) = \max(0,z)+1$ or $\text{hinge}(z) = \max(0,z+1)$ instead of $\text{softplus}(z)$.

Two essential properties of softplus are that: (a) it is convex° and (b) it upper bounds the step function. Note that srelu and hinge also enjoy these properties. Property (a) ensures that the optimization problem is relatively easy — under mild conditions, gradient descent will find a global minimum. By property (b), the total loss on a training set upper bounds the rate of erroneous classification on that training set. So loss is a *surrogate* for (in)accuracy: if the minimized loss is nearly zero, then the training accuracy is nearly 100%.°

← A function is **convex** when its graph is bowl-shaped rather than wriggly. It's easy to minimize convex functions by 'rolling downhill', since we'll never get stuck in a local wriggle. Don't worry about remembering or understanding this word.

← The perceptron satisfies (b) in a trivial way that yields a vacuous bound of 100% on the error rate.

So we have a family of related models: **logistic**, **perceptron**, and **SVM**. In Project 1 we'll find hypotheses optimal with respect to the perceptron and SVM models (the latter under a historical name of **pegasos**), but soon we'll focus mainly on logistic models, since they fit best with deep learning.

DEFINE NOTION OF LOSS!

RICHER OUTPUTS: MULTIPLE CLASSES — We've explored hypotheses $f_W(x) = \text{readout}(W\cdot\text{featurize}(x))$ where W represents the linear-combination step we tune to data. We began with **hard binary classification**, wherein we map

inputs to definite labels (say, $y = \text{cow}$ or $y = \text{dog}$):

$$\text{readout}(\mathfrak{d}) = \text{“cow if } 0 < \mathfrak{d} \text{ else dog”}$$

We then made this probabilistic using σ. In such **soft binary classification** we return (for each given input) a *distribution* over labels:

$$\text{readout}(\mathfrak{d}) = \text{“chance } \sigma(\mathfrak{d}) \text{ of cow; chance } 1-\sigma(\mathfrak{d}) \text{ of dog”}$$

Remembering that $\sigma(\mathfrak{d}) : (1 - \sigma(\mathfrak{d}))$ are in the ratio $\exp(\mathfrak{d}) : 1$, we rewrite:

$$\text{readout}(\mathfrak{d}) = \text{“chance of cow is } \exp(\mathfrak{d})/Z_{\mathfrak{d}}\text{; of dog, } \exp(0)/Z_{\mathfrak{d}}\text{”}$$

I hope some of you felt bugged by the above formulas' asymmetry: W measures "cow-ishness minus dog-ishness" — why not the other way around? Let's describe the same set of hypotheses but in a more symmetrical way. A common theme in mathematical problem solving is to trade irredundancy for symmetry (or vice versa). So let's posit both a W_{cow} *and* a W_{dog}. One measures "cow-ishness"; the other, "dog-ishness". They assemble to give W, which is now a matrix of shape $2 \times$number-of-features. So $\mathfrak{d}$ is now a list of 2 numbers: $\mathfrak{d}_{\text{cow}}$ and $\mathfrak{d}_{\text{dog}}$. Now $\mathfrak{d}_{\text{cow}} - \mathfrak{d}_{\text{dog}}$ plays the role that $\mathfrak{d}$ used to play.

Then we can do hard classification by:

$$\text{readout}(\mathfrak{d}) = \text{argmax}_y \mathfrak{d}_y$$

and soft classification by:

$$\text{readout}(\mathfrak{d}) = \text{“chance of } y \text{ is } \exp(\mathfrak{d}_y)/Z_{\mathfrak{d}}\text{”}$$

softmax plot

To make probabilities add to one, we divide by $Z_{\mathfrak{d}} = \sum_y \exp(\mathfrak{d}_y)$.

Behold! By rewriting our soft and hard hypotheses for binary classification, we've found formulas that also make sense for more than two classes! The above readout for **soft multi-class classification** is called **softmax**.

RICHER OUTPUTS: BEYOND CLASSIFICATION — By the way, if we're trying to predict a real-valued output instead of a binary label — this is called **hard one-output regression** — we can simply return $\mathfrak{d}$ itself as our readout: VERY OPTIONAL

$$\text{readout}(\mathfrak{d}) = \mathfrak{d}$$

softmax plot

This is far from the only choice! For example, if we know that the true ys will always be positive, then $\text{readout}(\mathfrak{d}) = \exp(\mathfrak{d})$ may make more sense. I've encountered a learning task (about alternating current in power lines) where what domain knowledge suggested — and what ended up working best — were trigonometric functions for featurization and readout! There are also many ways to return a distribution

instead of a number. One way to do such **soft one-output regression** is to use normal distributions:°

← Ask on the forum about the world of alternatives and how they influence learning!

$$\text{readout}(\mathfrak{d}) = \text{“normal distribution with mean } \mathfrak{d} \text{ and variance 25”}$$

Or we could allow for different variances by making $\mathfrak{d}$ two-dimensional and saying $\cdots$ mean $\mathfrak{d}_0$ and variance $\exp(\mathfrak{d}_1)$. By now we know how to do **multi-output regression**, soft or hard: just promote W to a matrix with more output dimensions.

TODO: show pictures of 3 classes, 4 classes (e.g. digits 0,1,8,9)

Okay, so now we know how to use our methods to predict discrete labels or real numbers. But what if we want to output structured data like text? A useful principle is to factor the task of generating such “variable-length” data into many smaller, simpler predictions, each potentially depending on what’s generated so far. For example, instead of using W to tell us how to go from (features of) an image x to a whole string y of characters, we can use W to tell us, based on an image x together with a partial string y', either what the next character is OR that the string should end. So if there are 27 possible characters (letters and space) then this is a $(27+1)$-way classification problem:

$$(\text{Images} \times \text{Strings}) \to \mathbb{R}^{\cdots} \to \mathbb{R}^{28} \to \text{DistributionsOn}(\{\text{'a'}, \cdots, \text{'z'}, \text{' '}, \text{STOP}\})$$

We could implement this function as some hand-crafted featurization function from Images $\times$ Strings to fixed-length vectors, followed by a learned W, followed by softmax.

Food For Thought: A “phylogenetic tree” is something that looks like

```
(dog.5mya.(cow.2mya.raccoon))
```

or

```
((chicken.63mya.snake).64mya.(cow)).120mya.(snail.1mya.slug)
```

That is, a tree is either a pair of trees together with a real number OR a species name. The numbers represent how long ago various clades diverged. Propose an architecture that, given a list of species, predicts a phylogenetic tree for that species. Don’t worry about featurization.

iterative optimization

By the end of this section, you’ll be able to

- implement gradient descent for any given loss function and (usually) thereby automatically and efficiently find nearly-optimal linear hypotheses from data
- explain why the gradient-update formulas for common linear models are sensible, not just formally but also intuitively

(STOCHASTIC) GRADIENT DESCENT — We seek a hypothesis that is best (among a class $\mathcal{H}$) according to some notion of how well each hypothesis models given data:

```
def badness(h,y,x):
    # return e.g. whether h misclassifies y,x OR h's surprise at seeing y,x OR etc
def badness_on_dataset(h, examples):
    return np.mean([badness(h,y,x) for y,x in examples])
```

Earlier we found a nearly best candidate by brute-force search over all hypotheses. But this doesn't scale to most interesting cases wherein $\mathcal{H}$ is intractably large. So: *what's a faster algorithm to find a nearly best candidate?*

A common idea is to start arbitrarily with some $h_0 \in \mathcal{H}$ and repeatedly improve to get $h_1, h_2, \cdots$. We eventually stop, say at h_{10000}. The key question is:° *how do we compute an improved hypothesis* h_{t+1} *from our current hypothesis* h_t?

← Also important are the questions of where to start and when to stop. But have patience! We'll discuss these later.

We *could* just keep randomly nudging h_t until we hit on an improvement; then we define h_{t+1} as that improvement. Though this sometimes works surprisingly well,° we can often save time by exploiting more available information. Specifically, we can inspect h_t's inadequacies to inform our proposal h_{t+1}. Intuitively, if h_t misclassifies a particular $(x_i, y_i) \in \mathcal{S}$, then we'd like h_{t+1} to be like h_t but nudged toward accurately classifying (x_i, y_i).°

← If you're curious, search 'metropolis hastings' and 'probabilistic programming'.

← In doing better on the ith datapoint, we might mess up how we do on the other datapoints! We'll consider this in due time.

How do we compute "a nudge toward accurately classifying (x, y)"? That is, how do measure how slightly changing a parameter affects some result? Answer: derivatives! To make h less bad on an example (y, x), we'll nudge h in tiny bit along $-g = -\texttt{dbadness}(h, y, x)/dh$. Say, h becomes $h - 0.01g$.° Once we write

```
def gradient_badness(h,y,x):
    # returns the derivative of badness(h,y,x) with respect to h
def gradient_badness_on_dataset(h, examples):
    return np.mean([gradient_badness(h,y,x) for y,x in examples])
```

← E.g. if each h is a vector and we've chosen $\texttt{badness}(h, y, x) = -yh \cdot x$ as our notion of badness, then $-\texttt{dbadness}(h, y, x)/dh = +yx$, so we'll nudge h in the direction of $+yx$.
Food For Thought: Is this update familiar?

we can repeatedly nudge via **gradient descent (GD)**, the engine of ML:°

```
h = initialize()
for t in range(10000):
  h = h - 0.01 * gradient_badness_on_dataset(h, examples)
```

← Food For Thought: Can GD directly minimize misclassification rate?

Since the derivative of total badness depends on all the training data, looping 10000 times is expensive. So in practice we estimate the needed derivative based on some *subset* (jargon: **batch**) of the training data — a different subset each pass through the loop — in what's called **stochastic gradient descent (SGD)**:

cartoon of GD

```
h = initialize()
for t in range(10000):
  batch = select_subset_of(examples)
  h = h - 0.01 * gradient_badness(h, batch)
```

(S)GD requires informative derivatives. Misclassification rate has uninformative derivatives: any tiny change in h won't change the predicted labels. But when we use probabilistic models, small changes in h can lead to small changes in the predicted *distribution* over labels. To speak poetically: the softness of probabilistic models paves a smooth ramp over the intractably black-and-white cliffs of 'right' or 'wrong'. We now apply SGD to maximizing probabilities.

cartoon of GD

MAXIMUM LIKELIHOOD ESTIMATION — When we can compute each hypothesis h's asserted probability that the training ys match the training xs, it seems reasonable to seek an h for which this probability is maximal. This method is **maximum likelihood estimation (MLE)**. It's convenient for the overall goodness to be a sum (or average) over each training example. But independent chances multiply rather than add: rolling snake-eyes has chance $1/6 \cdot 1/6$, not $1/6 + 1/6$. So we prefer to think about maximizing log-probabilities instead of maximizing probabilities — it's the same in the end.° By historical convention we like to minimize badness rather than maximize goodness, so we'll use SGD to *minimize negative-log-probabilities*.

← Throughout this course we make a crucial assumption that our training examples are independent from each other.

```
def badness(h,y,x):
    return -np.log( probability_model(y,x,h) )
```

Let's see this in action for the linear logistic model we developed for soft binary classification. A hypothesis $\vec{w}$ predicts that a (featurized) input $\vec{x}$ has label $y = +1$ or $y = -1$ with chance $\sigma(+\vec{w} \cdot \vec{x})$ or $\sigma(-\vec{w} \cdot \vec{x})$:

$$p_{y|x,w}(y|\vec{x}, \vec{w}) = \sigma(y\vec{w} \cdot \vec{x}) \qquad \text{where} \qquad \sigma(\mathfrak{d}) = 1/(1 - \exp(-\mathfrak{d}))$$

So MLE with our logistic model means finding $\vec{w}$ that *minimizes*

$$-\log(\text{prob of all } y_i\text{s given all } \vec{x}_i\text{s and } \vec{w}) = \sum_i -\log(\sigma(y_i \vec{w} \cdot \vec{x}_i))$$

The key computation is the derivative of those badness terms:°

← Remember that $\sigma'(z) = \sigma(z)\sigma(-z)$. To reduce clutter we'll temporarily write $y\vec{w} \cdot \vec{x}$ as ywx.

$$\frac{\partial(-\log(\sigma(ywx)))}{\partial w} = \frac{-\sigma(ywx)\sigma(-ywx)yx}{\sigma(ywx)} = -\sigma(-ywx)yx$$

Food For Thought: If you're like me, you might've zoned out by now. But this stuff is important, especially for deep learning! So please graph the above expressions to convince yourself that our formula for derivative makes sense visually.

To summarize, we've found the loss gradient for the logistic model:

```
sigma = lambda z : 1./(1+np.exp(-z))
def badness(w,y,x):             return -np.log( sigma(y*w.dot(x)) )
def gradient_badness(w,y,x):    return -sigma(-y*w.dot(x)) * y*x
```

As before, we define overall badness on a dataset as an average badness over examples; and for simplicity, let's intialize gradient descent at $h_0 = 0$:

```
def gradient_badness_on_dataset(h, examples):
  return np.mean([gradient_badness(h,y,x) for y,x in examples])
def initialize():
    return np.zeros(NUMBER_OF_DIMENSIONS, dtype=np.float32)
```

show trajectory in weight space over time – see how certainty degree of freedom is no longer redundant? ("markov")

Then we can finally write gradient descent:

```
h = initialize()
for t in range(10000):
  h = h - 0.01 * gradient_badness_on_data(h, examples)
```

show training and testing loss and acc over time

INITIALIZATION, LEARNING RATE, LOCAL MINIMA — VERY OPTIONAL

PICTURES OF TRAINING: NOISE AND CURVATURE — VERY OPTIONAL

test vs train curves: overfitting
random featurization: double descent

PRACTICAL IMPLEMENTATION: VECTORIZATION —

priors and generalization

By the end of this section, you'll be able to

- implement gradient descent for any given loss function and (usually) thereby automatically and efficiently find nearly-optimal linear hypotheses from data

(STOCHASTIC) GRADIENT DESCENT — We seek a hypothesis that is best (among a class $\mathcal{H}$) according to some notion of how well each hypothesis models given data:

```
def badness(h,y,x):
    # return e.g. whether h misclassifies y,x OR h's surprise at seeing y,x OR etc
def badness_on_dataset(h, examples):
    return np.mean([badness(h,y,x) for y,x in examples])
```

Earlier we found a nearly best candidate by brute-force search over all hypotheses. But this doesn't scale to most interesting cases wherein $\mathcal{H}$ is intractably large. So: *what's a faster algorithm to find a nearly best candidate?*

A common idea is to start arbitrarily with some $h_0 \in \mathcal{H}$ and repeatedly improve to get $h_1, h_2, \cdots$. We eventually stop, say at h_{10000}. The key question is:° *how do we compute an improved hypothesis* h_{t+1} *from our current hypothesis* h_t?

← Also important are the questions of where to start and when to stop. But have patience! We'll discuss these later.

We *could* just keep randomly nudging h_t until we hit on an improvement; then we define h_{t+1} as that improvement. Though this sometimes works surprisingly well,° we can often save time by exploiting more available information. Specifically, we can inspect h_t's inadequacies to inform our proposal h_{t+1}. Intuitively, if h_t misclassifies a particular $(x_i, y_i) \in \mathcal{S}$, then we'd like h_{t+1} to be like h_t but nudged toward accurately classifying (x_i, y_i).°

← If you're curious, search 'metropolis hastings' and 'probabilistic programming'.

How do we compute "a nudge toward accurately classifying (x, y)"? That is, how do measure how slightly changing a parameter affects some result? Answer: derivatives! To make h less bad on an example (y, x), we'll nudge h in tiny bit along $-g = -\texttt{dbadness}(h, y, x)/dh$. Say, h becomes $h - 0.01g$.° Once we write

← In doing better on the ith datapoint, we might mess up how we do on the other datapoints! We'll consider this in due time.

```
def gradient_badness(h,y,x):
    # returns the derivative of badness(h,y,x) with respect to h
def gradient_badness_on_dataset(h, examples):
    return np.mean([gradient_badness(h,y,x) for y,x in examples])
```

we can repeatedly nudge via **gradient descent (GD)**, the engine of ML:°

← E.g. if each h is a vector and we've chosen $\texttt{badness}(h, y, x) = -yh \cdot x$ as our notion of badness, then $-\texttt{dbadness}(h, y, x)/dh = +yx$, so we'll nudge h in the direction of $+yx$.
Food For Thought: Is this update familiar?

```
h = initialize()
for t in range(10000):
  h = h - 0.01 * gradient_badness_on_dataset(h, examples)
```

← Food For Thought: Can GD directly minimize misclassification rate?

Since the derivative of total badness depends on all the training data, looping 10000 times is expensive. So in practice we estimate the needed derivative based on some *subset* (jargon: **batch**) of the training data — a different subset each pass through the loop — in what's called **stochastic gradient descent (SGD)**:

```
h = initialize()
for t in range(10000):
  batch = select_subset_of(examples)
  h = h - 0.01 * gradient_badness(h, batch)
```

(S)GD requires informative derivatives. Misclassification rate has uninformative derivatives: any tiny change in h won't change the predicted labels. But when we use probabilistic models, small changes in h can lead to small changes in the predicted *distribution* over labels. To speak poetically: the softness of probabilistic models paves a smooth ramp over the intractably black-and-white cliffs of 'right' or 'wrong'. We now apply SGD to maximizing probabilities.

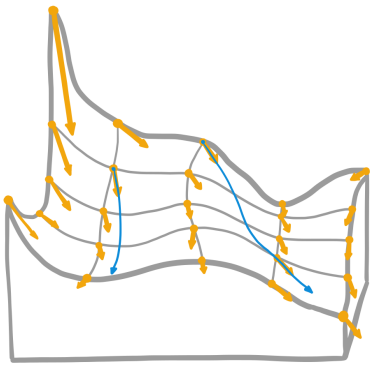


Figure 9: An intuitive picture of gradient descent. Vertical axis is training loss and the other two axes are weight-space. Warning: though logistic loss (and the L2 regularizer) is smooth, perceptron and hinge loss (and the L1 regularizer) have jagged angles or 'joints'. Also, all five of those functions are convex. The smooth, nonconvex picture here is most apt for deep learning (Unit 3).
cartoon of GD

MAXIMUM LIKELIHOOD ESTIMATION — When we can compute each hypothesis h's asserted probability that the training ys match the training xs, it seems reasonable to seek an h for which this probability is maximal. This method is **maximum likelihood estimation (MLE)**. It's convenient for the overall goodness to be a sum (or average) over each training example. But independent chances multiply rather than add: rolling snake-eyes has chance $1/6 \cdot 1/6$, not $1/6 + 1/6$. So we prefer to think about maximizing log-probabilities instead of maximizing probabilities — it's the same in the end.° By historical convention we like to minimize badness rather than maximize goodness, so we'll use SGD to *minimize negative-log-probabilities*.

← Throughout this course we make a crucial assumption that our training examples are independent from each other.

```
def badness(h,y,x):
    return -np.log( probability_model(y,x,h) )
```

Let's see this in action for the linear logistic model we developed for soft binary classification. A hypothesis $\vec{w}$ predicts that a (featurized) input $\vec{x}$ has label $y = +1$ or $y = -1$ with chance $\sigma(+\vec{w} \cdot \vec{x})$ or $\sigma(-\vec{w} \cdot \vec{x})$:

$$p_{y|x,w}(y|\vec{x}, \vec{w}) = \sigma(y\vec{w} \cdot \vec{x}) \qquad \text{where} \qquad \sigma(\mathfrak{d}) = 1/(1 - \exp(-\mathfrak{d}))$$

So MLE with our logistic model means finding $\vec{w}$ that *minimizes*

$$-\log(\text{prob of all } y_i\text{s given all } \vec{x}_i\text{s and } \vec{w}) = \sum_i -\log(\sigma(y_i \vec{w} \cdot \vec{x}_i))$$

The key computation is the derivative of those badness terms:°

$$\frac{\partial(-\log(\sigma(ywx)))}{\partial w} = \frac{-\sigma(ywx)\sigma(-ywx)yx}{\sigma(ywx)} = -\sigma(-ywx)yx$$

← Remember that $\sigma'(z) = \sigma(z)\sigma(-z)$. To reduce clutter we'll temporarily write $y\vec{w} \cdot \vec{x}$ as ywx.

Food For Thought: If you're like me, you might've zoned out by now. But this stuff is important, especially for deep learning! So please

graph the above expressions to convince yourself that our formula for derivative makes sense visually.

To summarize, we've found the loss gradient for the logistic model:

```
sigma = lambda z : 1./(1+np.exp(-z))
def badness(w,y,x):              return -np.log( sigma(y*w.dot(x)) )
def gradient_badness(w,y,x):     return -sigma(-y*w.dot(x)) * y*x
```

As before, we define overall badness on a dataset as an average badness over examples; and for simplicity, let's intialize gradient descent at $h_0 = 0$:

```
def gradient_badness_on_dataset(h, examples):
  return np.mean([gradient_badness(h,y,x) for y,x in examples])
def initialize():
    return np.zeros(NUMBER_OF_DIMENSIONS, dtype=np.float32)
```

Then we can finally write gradient descent:

```
h = initialize()
for t in range(10000):
  h = h - 0.01 * gradient_badness_on_data(h, examples)
```

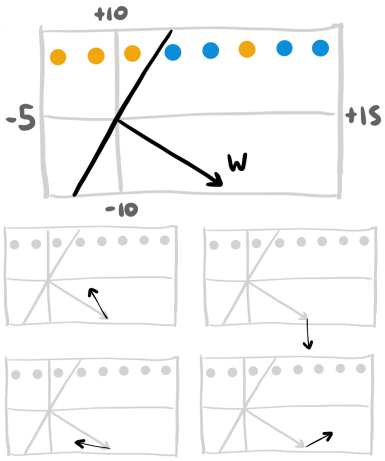


Figure 10: Food For Thought: Shown is a small training set (originally 1-D but featurized using the bias trick) and our current weight vector. Here blue is the positive label. *Which direction will the weight vector change if we do one step of (full-batch) gradient descent?* Four options are shown; one is correct. caption

show trajectory in weight space over time – see how certainty degree of freedom is no longer redundant? ("markov")
show training and testing loss and acc over time

LEAST-SQUARES REGRESSION — We've been focusing on classification. Regression is the same story. Instead of logistic probabilities we have gaussian probabilities. So we want to minimize° this loss (here, $\varphi(x)$ represents the feature vector for raw input x):

$$\lambda w \cdot w/2 + \sum_k (y_k - w \cdot \varphi(x_k))^2/2$$

instead of minimizing a classifier loss such as $\lambda w \cdot w/2 + \sum_k \log(1/\sigma(y_k w \cdot \varphi(x_k)))$.° We can do this by gradient descent. y-d heatmaps of logistic vs gaussian! contour maps showing that λ does not merely scale

However, in this case we can write down the answer in closed form. This gives us qualitative insight. (In practice our fastest way to evaluate that closed form formula *is* by fancy versions of gradient descent!) We want°

$$0 = \lambda w - \sum_k (y_k - w \cdot \varphi(x_k))\varphi(x_k)$$

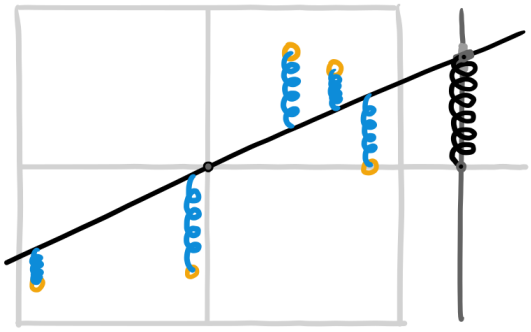
Since $(w \cdot \varphi(x_k))\varphi(x_k) = \varphi(x_k)\varphi(x_k)^T w$, we have

$$\sum_k y_k \varphi(x_k) = \left(\lambda \,\mathrm{Id} + \sum_k \varphi(x_k)\varphi(x_k)^T\right) w$$

or

$$w = \left(\lambda \,\mathrm{Id} + \sum_k \varphi(x_k)\varphi(x_k)^T\right)^{-1} \left(\sum_k y_k \varphi(x_k)\right)$$

Let's zoom way out:° neglecting that we can't divide by vectors and neglecting λ, we read the above as saying that $w = (x^2)^{-1}(yx) = y/x$; this looks right since w is our ideal exchange rate from x to y! The λ makes that denominator a bit bigger: $w = (\lambda + x^2)^{-1}(yx) = y/(x + \lambda/x)$. A bigger denominator means a smaller answer, so λ pulls w toward 0. Due to the xs in the denominator, λ has less effect — it pulls w less — for large xs. This is because large xs have 'more leverage', i.e. are more constraining evidence on what w ought to be.



Food For Thought: Momentarily neglect λ. Visualize one gradient update on the spring figure above: is the torque clockwise or anti-clockwise?

← Food For Thought: Suppose all the training points x_k are the same, say $(1, 0) \in \mathbb{R}^2$. What's the w optimal according to the above loss?

← Compare those two losses when $(y_k, w \cdot \varphi(x_k)) = (+1, +10)$ or $(-1, +10)$.

← Food For Thought: Verify that the gradient (with respect to w) of the above loss is

$$\lambda w - \sum_k (y_k - w \cdot \varphi(x_k))\varphi(x_k)$$

If $\lambda = 0$ and $w \cdot \varphi(x_k)$ overestimates y_k, which way would a gradient update on the kth training example change w? Is this intuitive?

← Food For Thought: Fix some training set. Consider the optimal weight $w_\star$ as a function of our regularization strength λ. How does the regularization value $\lambda w_\star \cdot w_\star/2$ change as we increase λ?

Figure 11: Here's an illuminating physical analogy. The least-squares loss says we want decision function values d to be close to the true ys. So we can imagine hooking up a stretchy spring from each training point (x_i, y_i) to our **predictor line (or hyperplane)**. Bolt that line to the origin, but let it rotate freely. The springs all want to have zero length. Then minimizing least-squares loss is the same as minimizing potential energy! We can also model L2 regularizers, which say that the predictor line wants to be horizontal. So we tie a **special (black) spring** from the input-space (say, at $x = 1$) to the line. The larger the L2's λ, the stiffer this spring is compared to the others. To get this to fully work, we need to keep all the springs vertical. (The mechanically inclined reader might enjoy imagining joining together a pair of slippery sheaths, one slipping along the predictor line and the other slipping along a fixed vertical pole.) Then the analogy is mathematically exact. TODO EXPAND ON THIS CAPTION!

Now, is there some value of λ for which the springs are in static equilibrium?

Food For Thought: True or false: the most stretched-out (blue) spring contributes the greatest non-regularizer term to the loss gradient?

INITIALIZATION, LEARNING RATE, LOCAL MINIMA — VERY OPTIONAL

PICTURES OF TRAINING: NOISE AND CURVATURE — VERY OPTIONAL

test vs train curves: overfitting
random featurization: double descent

PRACTICAL IMPLEMENTATION: VECTORIZATION —

model selection

C. bend those lines to capture rich patterns (units 2,3)

featurization

FEATURES AS PRE-PROCESSING —

SKETCHING — VERY OPTIONAL

DOUBLE DESCENT — VERY OPTIONAL

ABSTRACTING TO DOT PRODUCTS — VERY OPTIONAL

KERNELIZED CLASSIFIERS —

```
return 3 if condition(x) else 1
```

learned featurizations

IMAGINING THE SPACE OF FEATURE TUPLES — We'll focus on an architecture of the form

$$\hat{p}(y{=}{+}1\,|\,x) \;=\; (\sigma_{1\times 1}\circ A_{1\times(h+1)}\circ f_{(h+1)\times h}\circ B_{h\times d})(x)$$

where A, B are linear maps with the specified (input $\times$ output) dimensions, where σ is the familiar sigmoid operation, and where f applies the leaky relu function elementwise and concatenates a 1:

$$f((v_i : 0 \le i < h)) = (1,) + (\mathrm{lrelu}(v_i) : 0 \le i < h) \qquad \mathrm{lrelu}(z) = \max(z/10, z)$$

We call h the **hidden dimension** of the model. Intuitively, $f \circ B$ re-featurizes the input to a form more linearly separable (by weight vector A).

THE FEATURIZATION LAYER'S LEARNING SIGNAL —

EXPRESSIVITY AND LOCAL MINIMA —

"REPRESENTER THEOREM" —

locality and symmetry in architecture

2. multiple layers

We can continue alternating learned linearities with fixed nonlin-

earities:

$$\hat{p}(y=+1\,|\,x) = (\sigma_{1\times 1} \circ A_{1\times(h''+1)} \circ f_{(h''+1)\times h''} \circ B_{h''\times(h'+1)} \circ f_{(h'+1)\times h'} \circ C_{h'\times(h+1)} \circ f_{(h+1)\times h} \circ D_{h\times d})(x)$$

FEATURE HIERARCHIES —

BOTTLENECKING —

HIGHWAYS —

3. architecture and wishful thinking

REPRESENTATION LEARNING —

4. architecture and symmetry

dependencies in architecture

5. stochastic gradient descent

6. loss landscape shape

D. thicken those lines to quantify uncertainty (unit 4)

bayesian models

examples of bayesian models

inference algorithms for bayesian models

combining with deep learning

E. beyond learning-from-examples (unit 5)

reinforcement ; bandits

state (dependence on prev xs and on actions) ; RL ; partial observations

deep q learning

learning-from-instructions ; farewell

About to speak at [conference]. Spilled Coke on left leg of jeans, so poured some water on right leg so looks like the denim fade.
— tony hsieh

The key to success is failure.
— michael j. jordan

The virtue of maps, they show what can be done with limited space, they foresee that everything can happen therein.
— josé saramago