Basics of Machine Learning

CONTRIBUTORS — karene, laurent, sam,
append your name to the list above by editing 'CONTRIBUTORS.tex'

HALLO! — These optional and still-evolving notes for 6.86x give a big picture view of our conceptual landscape. They might help you study, but you can do all your assignments without these notes. These notes treat many but not all the topics of our course; they also treat a few topics not emphasized in class, to fill gaps for curious learners.

I'm happy to answer questions about the notes on the forum. If you want to help improve these notes, ask me; I'll be happy to list you as a contributor!

We compiled these notes using the following color palette; let us know if the colors are hard to distinguish.

NAVIGATING THESE NOTES — We divide these notes into bite-size passages; each begins with a gray heading (e.g. 'navigating these notes' to the left). All passages are optional in the sense that you'll be able to complete all assignments without ever studying these notes. Yet, aside from a few bonus passages marked as VERY OPTIONAL, most passages offer complementary discussion of our course topics that may help you understand, appreciate, and internalize the lectures. After all, only with two lines of sight can we see depth.

These notes highlight probability theory and model architecture more than the lectures; clustering and low-rank approximation, less. We can use different projections to tear and squash a globe into a flat map of earth. Different views reveal different aspects of geography. In order to "keep australia from being split into two parts" and to "display the south pole", I used a different map projection than the lectures do. So we traverse topics in the notes in a different order than we do in lecture. Still, we cover essentially the same overall ground.

CLICKABLE TABLE OF CONTENTS

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.

Before getting into *'how machines actually "learn" things'*, we must realize that machines and humans do not learn and understand things the same way. Perhaps you have wondered why on some websites, we are asked to solve a simple image CAPTCHA!? Because if we can solve them, then wouldn't it be easier for any machine to solve such a problem?° But, despite what you think, "understanding" what is in the image still remains one of the most challenging tasks for a machine to learn!

← The state-of-the-art machines/computers today are capable of performing billions of billions (quintillions) of floating-point operations per second!

It is safe to assume that even now, nobody completely understands how a human mind works, and because of the lack of this insight, we struggle to teach machines how to actually "think" or learn. Significant advancements in the field of machine learning and artificial intelligence have happened recently, but I still believe we are quite far from achieving "*true intelligence*" in machines.

Then, how should we approach such a difficult task of teaching machines to do something intelligent? Fortunately, not all hope is lost! We have several ways of teaching machines how to do specific works efficiently and intelligently (I still feel some discomfort when it comes to using the word 'intelligent' for machines.).

KINDS OF LEARNING — So, what are those different ways of learning that we can use to teach machines? We can teach them:

by instruction: "to know 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 methods. We'll focus on the most important method: **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".

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.

← 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*.

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

← 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.

In practice we'll think of each hypothesis as mapping mushrooms to *distribu-*

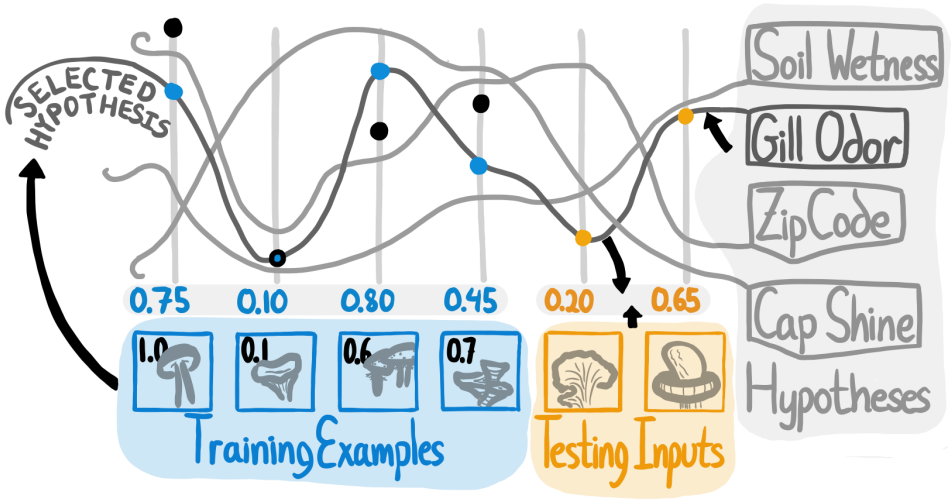


Figure 1: **Predicting mushrooms' poisons.** Our learning program selects from a class of hypotheses (gray blob) a plausible hypothesis that well fits (**blue dots** are close to **black dots**) a given list of poison-labeled mushrooms (blue blob). Evaluating the selected hypothesis on new mushrooms,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

tions 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*.°

← 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.

a tiny example: classifying handwritten digits

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 $x \in \mathcal{X}$.

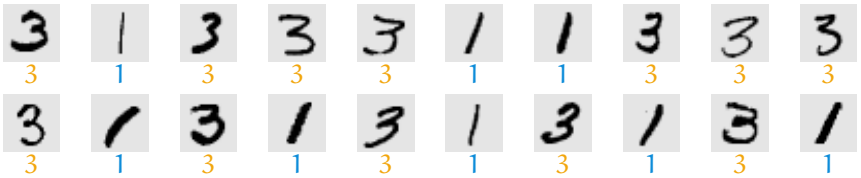
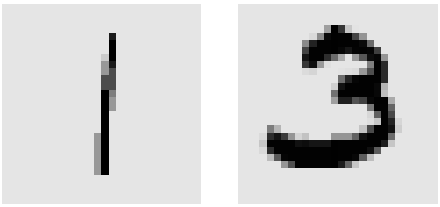


Figure 2: Twenty example pairs. Each photo x is a 28×28 grid of numbers representing pixel intensities. The light gray background has intensity 0.0; the blackest pixels, intensity 1.0. Below each photo x we display the corresponding label y: either $y = 1$ or $y = 3$. We'll adhere to this color code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

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.])
```

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
```

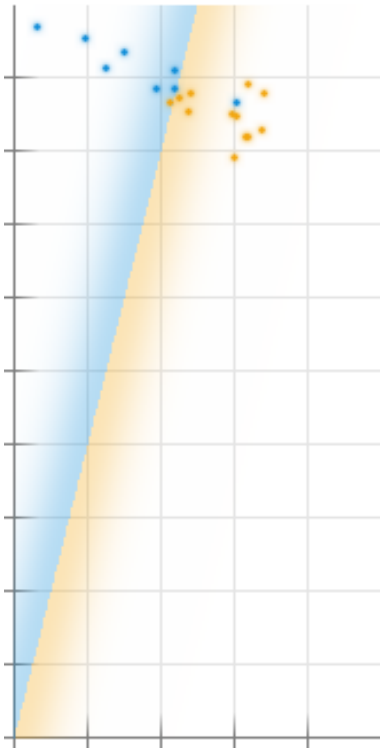


Figure 3: **Featurized training data.** Our $N = 20$ many training examples, viewed in the brightness-width plane. The vertical *brightness* axis ranges $[0.0, 1.0]$; the horizontal *width* axis ranges $[0.0, 0.5]$. The origin is at the lower left. Orange dots represent $y = 3$ examples; blue dot, $y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

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)?

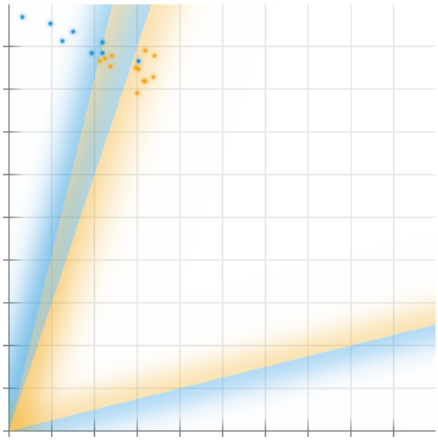


Figure 4: **Hypotheses differ in training accuracy: feature space.** 3 hypotheses classify training data in the brightness-width plane (axes range $[0, 1.0]$). Glowing colors distinguish a hypothesis' 1 and 3 sides. For instance, the bottom-most line classifies all the training points as 3s. **Caution**: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

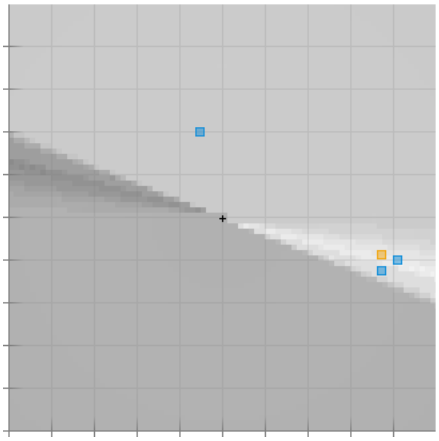


Figure 5: **Hypotheses differ in training accuracy: weight space.** We visualize $\mathcal{H}$ as the (a, b)-plane (axes range $[-99, +99]$). Each point determines a whole line in the brightness-width plane. Shading shows training error: darker points misclassify more training examples. The least shaded, 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?

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.

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? analyzing our error

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.

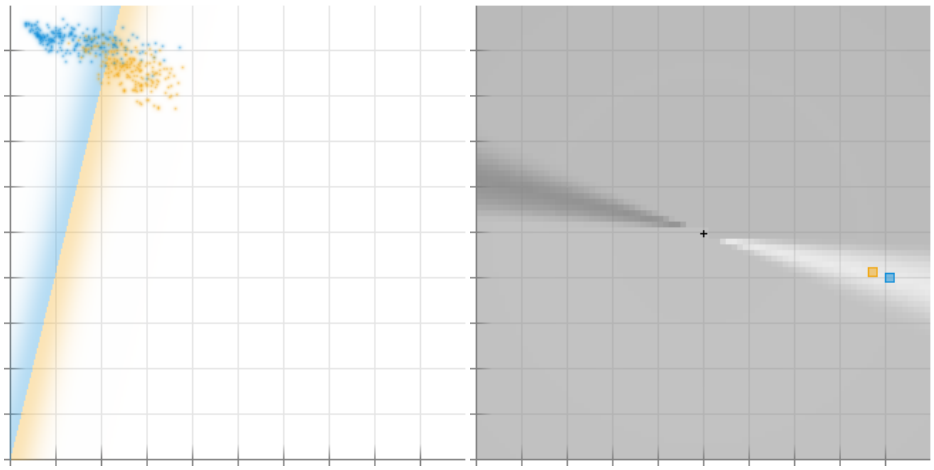


Figure 6: **Testing error visualized two ways.**. — **Left: in feature space.** The hypotheses $h = (20, 83)$ that we selected based on the training set classifies testing data in the brightness-width plane; glowing colors distinguish a hypothesis' 1 and 3 sides. Axes range $[0, 1.0]$. — **Right: in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

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. 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. VERY OPTIONAL 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 $\text{tst}(f)$.

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)

Learning from Examples

goodness of fit

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

- define (in both math and code) a class of linear hypotheses for given featurized data
- compute the perceptron and hinge losses a given hypothesis suffers on given data

WHAT HYPOTHESES WILL WE CONSIDER? A RECIPE FOR $\mathcal{H}$ — Remember: we want our machine to find an input-to-output rule. We call such rules **hypotheses**. As engineers, we carve out a menu $\mathcal{H}$ of rules for the machine to choose from. We'll consider hypotheses of this format: *extract features* of the input to make a list of numbers; then *linearly combine* those numbers to make another list of numbers; finally, *read out* a prediction from the latter list. Our digit classifying hypotheses, for instance, look like:°

```
def predict(x):
  features = [brightness(x), width(x)]              # featurize
  threeness = [ -1.*features[0] +4.*features[1] ]   # linearly combine
  prediction = 3 if threeness[0]>0. else 1          # read out prediction
  return prediction
```

← Our list `threeness` has length one: it's just a fancy way of talking about a single number. We'll later use longer lists to model richer outputs: to classify between > 2 labels, to generate a whole image instead of a class label, etc. The concept of `threeness` (and what plays the same role in other ML projects) is called the **decision function**. The decision function takes an input x and returns some kind of score for what the final output should be. The decision function value is what we feed into the readout.

The various hypotheses differ only in those coefficients (jargon: **weights**, here $-1, +4$) for their linear combinations; it is these degrees of freedom that the machine *learns* from data. These arrows summarize the situation:°

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

← This triple of arrows is standard in ML. But the name 'readout' is not standard jargon. I don't know standard jargon for this arrow.

Our Unit 1 motto is to *learn a linear map flanked by hand-coded nonlinearities*. We design the nonlinearities to capture domain knowledge about our data and goals.

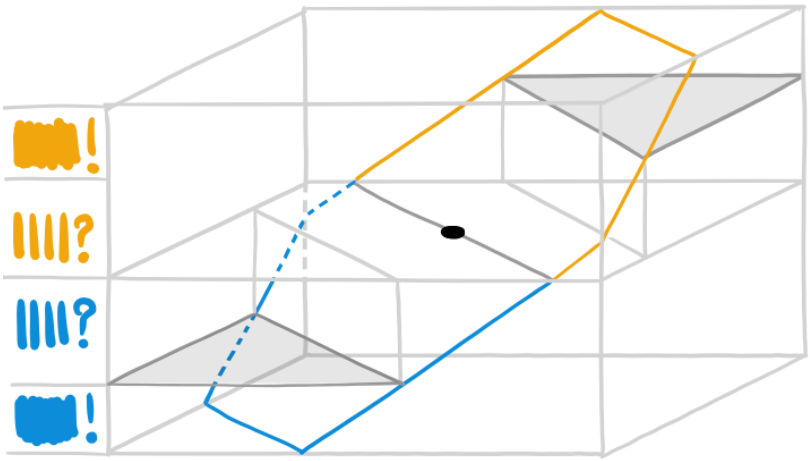


Figure 7: **The *slope* of the decision function's graph is meaningful, even though it doesn't affect the decision boundary.** We plot the decision function (vertical axis) over weight-space (horizontal axes) (**black dot**). The graph is an infinite plane, a hexagonal portion of which we show; it attains zero altitude at the decision boundary. We interpret high (low) altitude as 'confident the label is orange (blue)'; intermediate altitudes, as leaning but without confidence. We'll introduce a notion of 'goodness-of-fit-to-data' that seeks not just to correctly classify all the training points BUT ALSO to do so with confidence. That is, we want orange (blue) training points to all be above (below) the gray 'shelf' that delimits high-confidence. I like to imagine a shoreline: an interface between water and beach. We want to adjust the topography so that all blue (orange) training points are deep underwater (high and dry).

For now, we'll assume we've already decided on our featurization and we'll use the same readout as in the code above:°

```
prediction = 3 if threeness[0]>0. else 1          # for a digit-classifying project
prediction = 'cow' if bovinity[0]>0. else 'dog'   # for an animal-classifying project
```

That's how we make predictions from those "**decision function values**" `threeness` or `bovinity`. But to compute those values we need to determine some weights. *How shall we determine our weights and hence our hypothesis?*

caption paragraph flow?! ← Intuitively, `threeness` is the machine's **confidence** that the answer is 3. Our current readout discards all information except for the sign: we forget whether the machine was very confident or slightly confident that the answer is 3, so long as it prefers 3 over 1. However, we will care about confidence levels when assessing goodness-of-fit.

HOW GOOD IS A HYPOTHESIS? FIT — We instruct our machine to find within our menu $\mathcal{H}$ a hypothesis that's as "good" as possible. That is, the hypothesis should both fit our training data and seem intrinsically plausible. We want to quantify these notions of goodness-of-fit and intrinsic-plausibility. As with $\mathcal{H}$, how we quantify these notions is an engineering art informed by domain knowledge. Still, there are patterns and principles — we will study two specific quantitative notions, the **perceptron loss** and **SVM loss**, to study these principles. Later, once we understand these notions as quantifying uncertainty (i.e., as probabilistic notions), we'll appreciate their logic. But for now we'll bravely venture forth, ad hoc!

We start with goodness-of-fit. Hypotheses correspond° to weights. For example, the weight vector $(-1,+4)$ determines the hypothesis listed above.

One way to quantify h's goodness-of-fit to a training example (x,y) is to see whether or not h correctly predicts y from x. That is, we could quantify goodness-of-fit by *training accuracy*, like we did in the previous digits example:

```
def is_correct(x,y,a,b):
  threeness = a*brightness(x) + b*width(x)
  prediction = 3 if threeness>0. else 1
  return 1. if prediction==y else 0.
```

← A very careful reader might ask: *can't multiple choices of weights determine the same hypothesis?* E.g. $(-1,+4)$ and $(-10,+40)$ classify every input the same way, since they either both make `threeness` positive or both make `threeness` negative. This is a very good point, dear reader, but at this stage in the course, much too pedantic! Ask again later.

By historical convention we actually like° to minimize badness (jargon: **loss**) rather than maximize goodness. So we'll rewrite the above in terms of mistakes:°

```
def leeway_before_mistake(x,y,a,b):
  threeness = a*brightness(x) + b*width(x)
  return +threeness if y==3 else -threeness
def is_mistake(x,y,a,b):
  return 0. leeway_before_mistake(x,y,a,b)>0. else 1.
```

← ML can be a glass half-empty kind of subject!

← We'll refer to the "leeway before a mistake" throughout this section; it's a standard concept but **not** a concept with a standard name afaik.

We *could* define goodness-of-fit as training accuracy. But we'll enjoy better generalization and easier optimization by allowing "partial credit" for borderline predictions. E.g. we could use `leeway_before_mistake` as goodness-of-fit:°

```
def linear_loss(x,y,a,b):
  return 1 - leeway_before_mistake(x,y,a,b)
```

← to define *loss*, we flip signs, hence the '1−'

Food For Thought: For incentives to point the right way, loss should *decrease as* `threeness` *increases when* `y==3` but should *increase as* `threeness` *increases when* `y==1`. Verify these relations for the several loss functions we define.

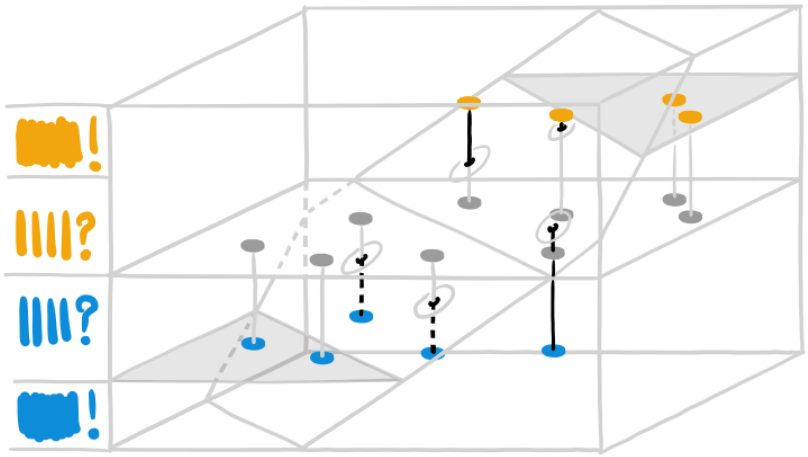


Figure 8: **Geometry of leeways, margins, and hinge loss**. As before, we graph the decision function (vertical axis) against weight-space. The $y=-1$ ($y=+1$) training points (x,y) sit one unit below (above) the zero-altitude plane. We draw regions where the graph surpasses ± 1 altitude as shaded 'shelves'. **Leeway** is the vertical distance between a gray x and the graph (and it's positive when the point is correctly classified). **Linear loss** is the vertical distance between an (x,y) point and the graph (positive when the point is on the 'wrong' side of the graph, i.e., when the graph lies between the point and the zero-altitude plane). **Hinge loss** is linear loss, except no reward is given for any distance beyond the 'shelves' at ± 1 altitude. We draw hinge loss in **black vertical segments**. A hypothesis's **margin** is the horizontal distance the edge of those 'shelves' and the decision boundary. caption

But, continuing the theme of pessimism, we usually feel that a "very safely classified" point (very positive `leeway`) shouldn't make up for a bunch of "slightly misclassified" points (slightly negative `leeway`).° But total linear loss doesn't capture this asymmetry; to address this, let's impose a floor on `linear_loss` so that it can't get too negative (i.e., so that positive `leeway` doesn't count arbitrarily much). We get **perceptron loss** if we set a floor of 1; **SVM loss** (also known as **hinge loss**) if we set a floor of 0:

```
def perceptron_loss(x,y,a,b):
  return max(1, 1 - leeway_before_mistake(x,y,a,b))
def svm_loss(x,y,a,b):
  return max(0, 1 - leeway_before_mistake(x,y,a,b))
```

Proportional weights have the same accuracies but different hinge losses.
Food For Thought: Can we say the same of perceptron loss?

← That is, we'd rather have `leeways` $+.1, +.1, +.1, +.1$ than $+10, -1, -1, -1$ on four training examples. A very positive `leeway` feels mildly pleasant to us while a very negative one feels urgently alarming.
Food For Thought: compute and compare the training accuracies in these two situations. As an open-ended followup, suggest reasons why optimizing `leeways` instead of just accuracies might help improve testing accuracy.

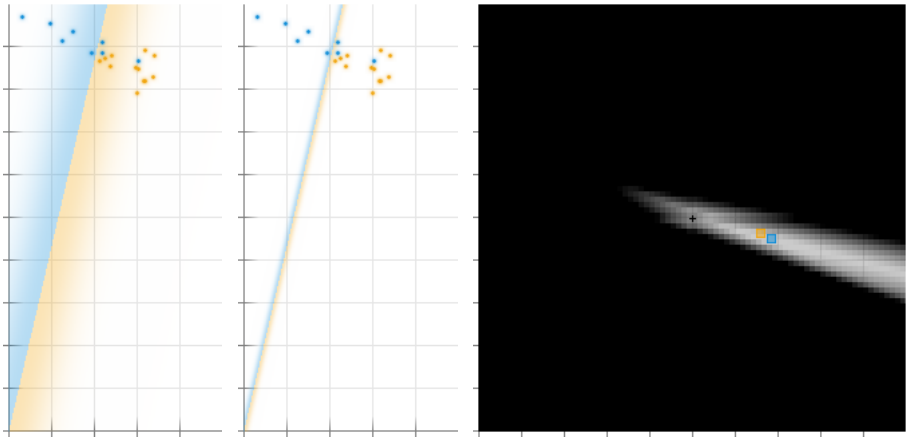


Figure 9: **Hinge loss's optimization landscape reflects confidence, unlike training accuracy's.** — **Left rectangular panes**. An under-confident and over-confident hypothesis. These have weights $(a/3, b/3)$ and $(3a, 3b)$, where $(a, b) = (-3.75, 16.25)$ minimizes training hinge loss. The glowing colors' width indicates how rapidly `leeway` changes as we move farther from the boundary. — **Right shaded box**. The (a, b) plane, centered at the origin and shaded by hinge loss, with training optimum blue. Indecisive hs (e.g. if `threeness`≈ 0) suffer, since $\max(0, 1-\ell) \approx 1$, (not 0) when $\ell \approx 0$. Hinge loss penalizes *overconfident* mistakes severely (e.g. when $y = 1$ yet `threeness` is huge): $\max(0, 1-\ell)$ is unbounded in ℓ. If we start at the origin $(a, b) = (0, 0)$ and shoot (to less underconfidence) toward the optimal hypothesis, loss will decrease; but once we pass that optimum (to overconfidence), loss will (slowly) start increasing again.

smarter optimization

By the end of this section, you'll be able to
- tackle minimize_perceptron or hinge loss via 'gradient' updates
- explain why those update formulas common linear models are intuitively sensible

WHICH HYPOTHESIS IS BEST? — Now that we've quantified badness-of-fit-to-data, we want to find a hypothesis $h = (a, b)$ that minimizes it.° We *could* try brute force, like so:

```
def best_hypothesis():
  # returns a pair (loss value, hypothesis)
  return min(perceptron_loss((training_data, a, b), (a,b))
             for a in np.arange(-50,+50,.25)
             for b in np.arange(-50,+50,.25)              )
```

But this is slow! Here we're searching a 2D grid at resolution ≈ 400, so we call the loss 400^2 times. That exponent counts the parameters we're finding (here, 2: a and b); if we had 10 features and 10 weights, we'd make 400^{10} calls. Yikes!

Let's instead use more of the information available to direct our search. Suppose at some point in our search the best h we've found so far is (a, b). The loss function is a sum (or average) over N training points (x_i, y_i):°

$$+\max(1, 1 - y_0(a \cdot \text{br}(x_0) + b \cdot \text{wi}(x_0)) + \cdots$$
$$+\max(1, 1 - y_{42}(a \cdot \text{br}(x_{42}) + b \cdot \text{wi}(x_{42})) + \cdots$$

← Soon we'll also include intrinsic implausibility! We'll see throughout this course that it's important to minimize implausibility plus badness-of-fit, not just badness-of-fit; otherwise, optimization might select a very implausible hypothesis that happens to fit the the training data. Think of the Greek constellations: isn't it miraculous how constellations — the bears, the queen, etc — so perfectly fit the positions of the stars?

← Here, $\text{br}(x)$ and $\text{wi}(x)$ stand for the features of x, say the brightness and width. Also, we'll use take y's values to be ± 1 (rather than cow vs dog or 1 vs 3), for notational convenience.

Let's try to decrease this sum by reducing one row at a time. If $\ell > 0$, then any small change in (a, b) won't change $\max(1, 1-\ell)$. But if $\ell \leq 0$, then we can decrease $\max(1, 1-\ell)$ by increasing ℓ, i.e., by increasing (say):

$$\underbrace{+1}_{y_{42}}(a \cdot \underbrace{0.9}_{br(x_{42})} + b \cdot \underbrace{0.1}_{wi(x_{42})})$$

We can increase ℓ by increasing a or b; but increasing a gives us more bang for our buck ($0.9 > 0.1$), so we'd probably nudge a more than b, say, by adding a multiple of $(+0.9, +0.1)$ to (a, b). Conversely, if $y_i = -1$ then we'd add a multiple of $(-0.9, -0.1)$ to (a, b). Therefore, to reduce the ith row, we want to move a, b like this: *Unless the max term is* 0*, add a multiple of* $y_i(br(x_i), wi(x_i))$ *to* (a, b).

Now, what if improving the ith row messes up other rows? Because of this danger we'll take small steps: we'll scale those aforementioned multiples by some small η. That way, even if the rows all pull (a, b) in different directions, the dance will buzz close to some average (a, b) that minimizes the average row. So let's initialize $h = (a, b)$ arbitrarily and take a bunch of small steps!

```
ETA = 0.01
h = initialize()
for t in range(10000):
  xfeatures, y = fetch_datapoint_from(training_examples)
  leeway = y*h.dot(xfeatures)
  h = h + ETA * ( y * xfeatures * (0 if leeway>0. else 1) ) # update
```

Food For Thought: Convince a friend that, for $\eta = \text{ETA} = 1$, this is the **perceptron algorithm** from lecture. Choosing smaller η means that it takes more steps to get near an optimal h but that once we get near we will stay nearby instead of jumping away. One can aim for the best of both worlds by letting η decay with t.
Food For Thought: We could have used hinge loss instead of perceptron loss. Mimicking the reasoning above, derive a corresponding line of code `h = h + ...`.

intrinsic plausibility

model selection

Engineering Features (and Friends) by Hand

featurization

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

- tailor a hypothesis class by designing features that reflect domain knowledge
- recognize the geometric patterns that common nonlinear featurizations help express

DESIGNING FEATURIZATIONS — Remember: our motto in Units 1 and 2 is to *learn linear maps flanked by hand-coded nonlinearities*. That is, we consider hypotheses of this format:

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

In this section and the next we'll design those non-learned functions — the featurizers and readouts, respectively — to construct a hypothesis class $\mathcal{H}$ suitable given our domain knowledge and our goals. In this section, we'll discuss how the design of features determines the patterns that the machine is able to express; feature design can thus make or break an ML project.

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 darknesses 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.

This bias trick is non-linear in that it shifts the origin. Fancier non-linearities enrich our vocabulary of hypotheses in fancier ways. Take a look these decision boundaries using a degree-2 monomial feature (left) and (right) a non-polynomial 'black hole' feature for two cartoon datasets:

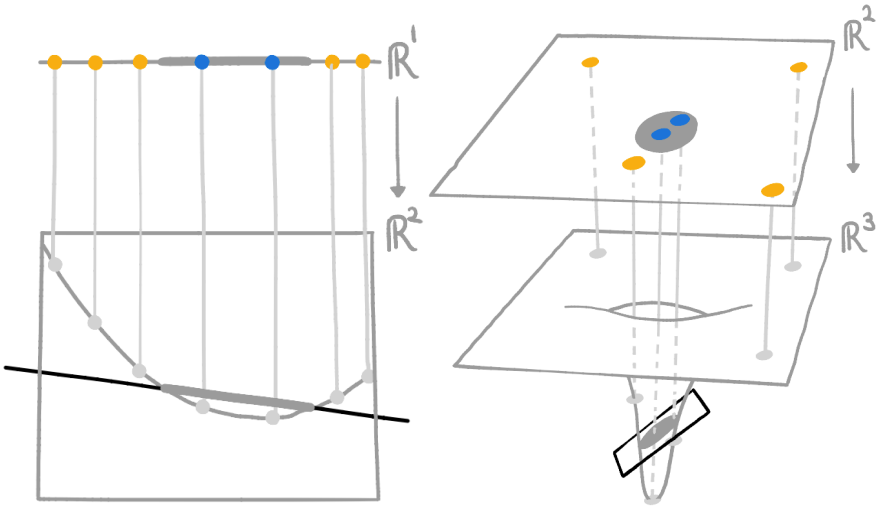


Figure 11: **Fancier nonlinearities help us model curvy patterns using linear weights. Left**: Using a quadratic feature (and the bias trick, too) we can learn to classify points in $\mathcal{X} = \mathbb{R}^1$ by whether they are in some *interval* **gray region**. We don't know the interval beforehand, but fortunately, different linear decision boundaries (**black lines**) in feature-space give rise to *all possible* intervals! We may optimize as usual to find the best interval.
Right: No line through $\mathcal{X} = \mathbb{R}^2$ separates our raw data. But a **(black) hyperplane** *does* separate our featurized data. We are thus able to learn a hypothesis that predicts the blue label for xs inside the **gray region**. Intuitively, the 'black hole' feature measures whether an input x is nearby a certain point in $\mathcal{X}$.
Food For Thought: Our 'black hole' decision boundary actually has two parts, one finite and one infinite. The infinite part is slightly 'off-screen'. Sketch the full situation out!

Say $\mathcal{X} = \mathbb{R}^2$ has as its two raw features the latitude of low-orbit satellite and the latitude of a ground-based receiver, measured in degrees-from-the-equator. So the satellite is in earth's north or southern hemisphere and likewise for the receiver. We believe that whether or not the two objects are in the *same* hemisphere — call this situation 'visible' — is highly relevant to our ultimate prediction task. Food For Thought: For practice, we'll first try to classify xs by visibility. Define a degree-2 monomial feature $\varphi : \mathcal{X} \to \mathbb{R}^1$ whose sign (± 1) tells us whether x is visible.° *Can direct linear use of* x*, even with the bias trick, predict the same?*
In reality we're interested in a more complex binary classification task on $\mathcal{X}$. For this we choose as features all monomials of degrees $0, 1, 2$ to get $\varphi : \mathcal{X} \to \mathbb{R}^6$.°
Food For Thought: Qualitatively describe which input-output rules $h : \mathcal{X} \to \{+1, -1\}$ we can express. For instance, which of these rules in the margin° can we express? (Formally, we can 'express' h if $h(x) = \text{sign}(w \cdot \varphi(x))$ for some w.)

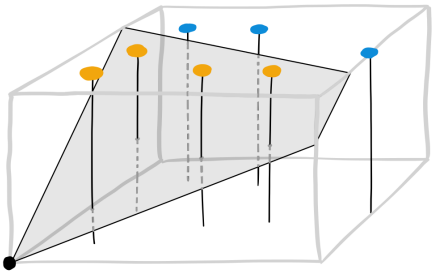
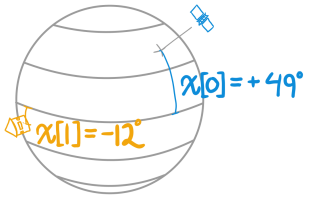


Figure 10: **The bias trick helps us model 'offset' decision boundaries.** Here, the origin is the lower right corner closer to the camera. Our raw inputs $x = (x[0], x[1])$ are 2-D; we can imagine them sitting on the bottom face of the plot (bottom ends of the vertical stems). But, within that face, 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.



← Thus, degree-2 monomials help model 2-way interactions between features.

← Food For Thought: φ maps to $\mathbb{R}^6$, i.e., we have exactly 6 monomials. Verify this!

← RULE A "+1 exactly when the satellite is between the Tropics of Cancer and of Capricorn" RULE B "+1 exactly when satellite and receiver are at the same latitude, up to 3° of error" RULE C "+1 exactly when both objects are above the Arctic Circle"

INTERPRETING WEIGHTS — The features we design ultimately get used according to a weight vector. So we stand to gain from deeper understanding of what weights 'mean'. Here we'll discuss three aspects of the 'intuitive logic' of weights.

First, **weights are not correlations**. A feature may correlate with a positive label (say, $y = +1$) yet fail to have a positive weight. That is: the *blah*-feature could correlate 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.

In Figure 12's center-left panel, the weight for brightness is negative even though both features positively correlate with blue! This is because brightness correlates *even better* with the *error* of soley-width-based prediction. So the optimal hypothesis, intuitively, uses the brightness as a 'correction' to width. This contrasts with the top-left panel, where the both correlations are still positive and both weights are positive. Intuitively, the optimal hypothesis here reduces noise by averaging a solely-brightness-based prediction with a solely-width-based one.

Second, **representing the same information two different ways can alter predictions**. A featurization doesn't just supply raw data: it also suggests to the machine which patterns are possible and, among those, which are plausible.

The bias trick and other nonlinear coordinate transforms illustrate this. But even *linear*, origin-preserving coordinate-transforms can alter predictions. For example, if we shear two features together — say, by using {preptime-plus-cooktime and cooktime} as features rather than {preptime and cooktime} — this can impact the decision boundary. 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 right three panels: the intersection of the two gray lines implicitly marks a testing point for which we predict different labels as we change coordinates. *Intuitively, the more stretched out a feature axis is, the more the learned hypothesis will rely on that feature.*°

Third, **features interact**. It's useful to think about 'the goodness' of individual features, but it's also good to realize that the reality is messier: a feature's predictive usefulness depends on which other features are present!

That is, a whole can be more (or less!) than the sum of its parts: the usefulness of a set of features ain't 'additive' the way the weight of a set of books is. Here's a simple example: let's flip two fair coins. We want to predict whether they ended up the same or not. Knowing the value of just the first coin is totally useless to prediction! Same with the value of just the second coin. BUT the two coins together help us predict 100% correctly.° Likewise, in the bottom-left panel the width is *independent* of the class label but not *conditionally-independent*-given-brightness. It's this that explains why we can't just compute the correlation of each feature with the predictand and then call it day.

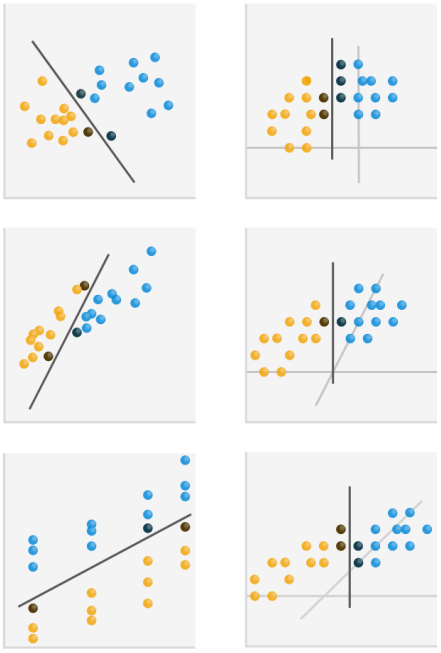


Figure 12: **Relations between feature statistics and optimal weights.** Each panel shows a different 2D binary classification task and a maximum-margin hypothesis. We shade margin-achieving points. To save ink we refer to the vertical and horizontal features as **brightness** and **width**; but you should be more imaginative. **Left:** *positive weights are consistent with positive, negative, or zero correlation!* **Right:** *presenting the same information in different coordinates (here, all 2D) alters predictions!*
Food For Thought: Think of classification tasks (and feature-pairs) that could plausibly give rise to the data depicted in each panel.

← Food For Thought: Explain the preceding intuition in terms of the L2 regularizer.

← Food For Thought: Find an analogue of the coins story where the whole is less than the sum of its parts, predictivity-wise.

richer outputs: quantifying uncertainty

By the end of this section, you'll be able to
- define a class of linear, probabilistic hypotheses appropriate to a given classification or regression task
- 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)

← 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.

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". Different probability models give different learning programs. In fact, two simple probability models give us the perceptron and hinge losses we saw previously!

	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	$\log_2(1+e^{(\cdot)})$	$\max(1,\cdot+1)$	$\max(0,\cdot+1)$
update	$1/(1+e^{+y\vartheta})$	step$(-y\vartheta)$	step$(1-y\vartheta)$

Table 1: **Three popular models for binary classification. Top rows:** Modeled chance given x that $y=+1,-1,\star$. We use $\vartheta=\vec{w}\cdot\vec{x}$, $\oplus=e^{+\vartheta/2}$, $\ominus=e^{-\vartheta/2}$, $a\wedge b=\min(a,b)$ 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

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

Food For Thought: For each of the three models, plot $p_{y|x}[+1|x]$ and $p_{y|x}[-1|x]$ against the decision function value ϑ. For which decision function values ϑ does the svm model allocate chance to the "no-comment" outcome $\star$? Relate to the margin.

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

Relate probabilities to losses for SVM, perceptron!

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 (relate to neg log likelihood)! PRIOR PROB as REGULARIZER!

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 $y=\text{dog}$):

$$\text{readout}(\vartheta)=\text{"cow if } 0<\vartheta \text{ else dog"}$$

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:

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

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**.

softmax plot
softmax plot

RICHER OUTPUTS: REGRESSION — 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:

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

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:

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

By now we know how to do **multi-output regression**, soft or hard: just promote W to a matrix with more output dimensions.

We can define the goodness-of-fit (of a probabilistic model to data) to be the log of the probability that model would have given to the data. Since log of a gaussian is quadratic in the distance between mean and data, our loss is quadratic in the same. But in our readout the mean is just the decision function value $\mathfrak{d}$. So minimizing loss means minimizing the squared distance $(\mathfrak{d} - y)^2$.

Food For Thought: We can avoid hardcoding constant variances by making $\mathfrak{d}$ two-dimensional and saying $\cdots$ mean $\mathfrak{d}_0$ and variance $\exp(\mathfrak{d}_1)$. Express loss in terms of the training data and the $2 \times$ (features) weight matrix W.

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

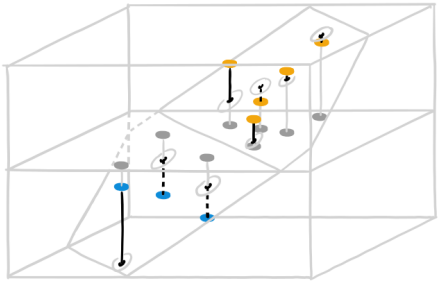


Figure 13: Minimizing loss means minimizing the squared distance $(\mathfrak{d} - y)^2$. We shade those distances **black**. Vertical axis: decision function value $\mathfrak{d}$. Horizontal axes: featurespace. Gray hexagon: (part of) the graph of the decision function $(x \mapsto w \cdot x = \mathfrak{d})$; the graph intersects $\mathfrak{d} = 0$ at the decision boundary (gray line). Inputs x are in dark gray; labeled pairs (x, y) are in orange and blue depending on y's sign. In regression, y can be any real number.

RICHER OUTPUTS: BEYOND CLASSIFICATION — 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 $(27+1+1)$-way classification problem:

VERY OPTIONAL

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

We could decide to 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. Deciding on this kind of thing is part of **architecture**.

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

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

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).°

← 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 $-g = -\texttt{dbadness}(h, y, x)/\texttt{dh}$. Say, h becomes $h - 0.01g$.° Once we write

← 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 $-\texttt{dbadness}(h, y, x)/\texttt{dh} = +yx$, so we'll nudge h in the direction of $+yx$.
Food For Thought: Is this update familiar?

```
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:°

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

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

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.

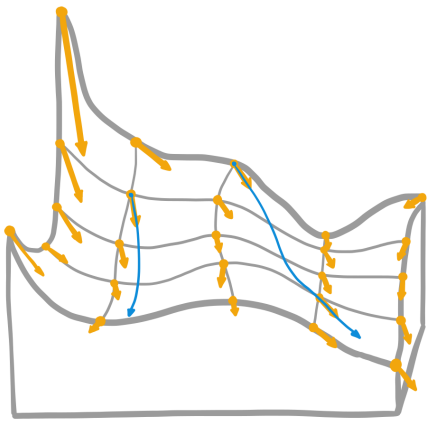


Figure 14: An intuitive picture of gradient descent. Vertical axis is training loss and the other two axes are weight-space. Warning: though logistic loss (and the L2 regularizer) is smooth, perceptron and hinge loss (and the L1 regularizer) have jagged angles or 'joints'. Also, all five of those functions 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)
```

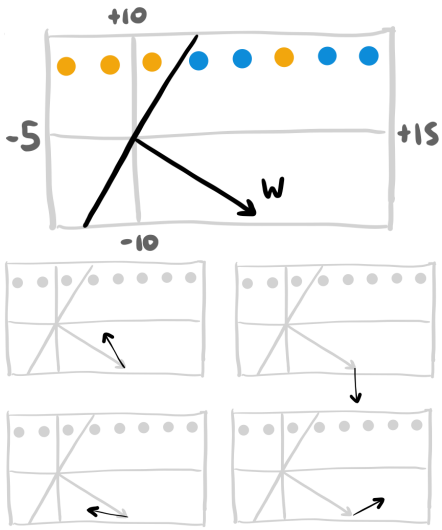


Figure 15: Food For Thought: Shown is a small training set (originally 1-D but featurized using the bias trick) and our current weight vector. Here blue is the positive label. *Which direction will the weight vector change if we do one step of (full-batch) gradient descent?* Four options are shown; one 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.

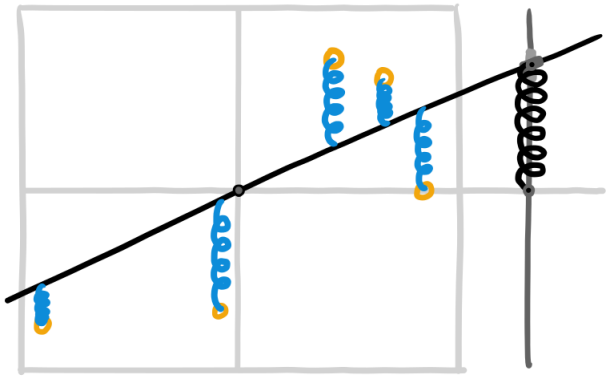


Figure 16: Here's an illuminating physical analogy. The least-squares loss says we want decision function values d to be close to the true ys. So we can imagine hooking up a stretchy spring from each training point (x_i, y_i) to our **predictor line (or hyperplane)**. Bolt that line to the origin, but 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!

Food For Thought: Momentarily neglect λ. Visualize one gradient update on the spring figure above: is the torque clockwise or anti-clockwise? Now, is there some value of λ for which the springs are in static equilibrium?

← 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?

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 —

data dependent features

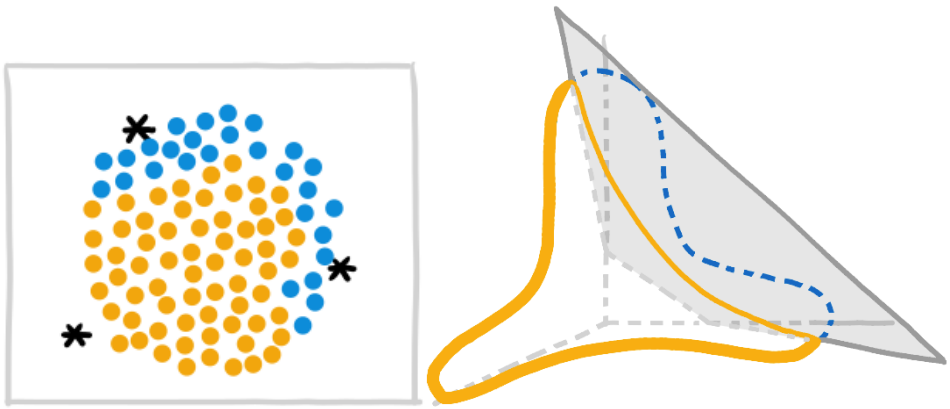
By the end of this section, you'll be able to
- featurize based on *similarity* and *low-rank approximations* to present data in a form that makes learning complex patterns easy
- use the kernel trick to quickly optimize very-high-dimensional models

LANDMARKS AND SIMILARITY — The 'black-hole' nonlinearity (Figure 17) nicely separates its training data. When domain knowledge says some 'landmark' in $\mathcal{X}$ is salient, we can make a 'black-hole-shaped' feature centered on that landmark. Intuitively, such a feature measures how *similar* a given x is to the landmark. (In general we define similarity using domain knowledge.)

If something works well, it's worth trying it more (`:p`), so let's now make multiple black-hole-shaped features, one centered around each of *several* landmarks! If we use three representative 'landmarks' $x_\circ, x_\square, x_\star$, say, we get a featurization

$$\varphi(x) = (\text{similarity}(x, x_\circ), \text{similarity}(x, x_\square), \text{similarity}(x, x_\star)) \in \mathbb{R}^3$$

Here's what the decision boundary might look like if $\mathcal{X} = \mathbb{R}^2$ and we define similarity(x, x') to be big when x, x' are closer than a centimeter in Euclidean distance, but then to rapidly decay as x, x' move apart. Our three **landmarks (black asterisks)** lie to the south-west, north, east. Our nonlinear φ gives us a curvy boundary in raw $\mathcal{X}$ space (left) from each **linear hyperplane** through featurespace (right). Intuitively, the hyperplane's weights say, "*predict orange unless x is much closer to the N and E landmarks than to the SW landmark*":



This looks nice. Very expressive.° *But how do we choose our landmarks?* That is: *what if we lack domain knowledge specific enough to tell us which reference-points in $\mathcal{X}$ are important?* Well, we might guess that some (we don't know which) of our N training points may be useful landmarks. Since we don't know which, let's center 'black-holes' at *every* training point at once! Then we get N new features

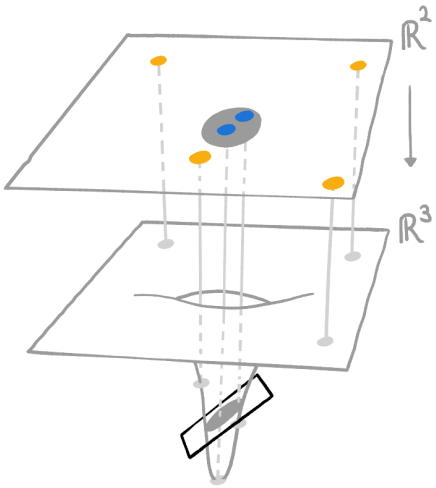
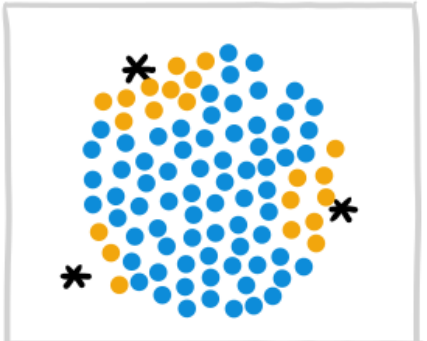


Figure 17: **A nonlinear 'black-hole' feature** (encountered previously). We map raw 2-D inputs x to 3-D feature-vectors $\varphi(x)$s, defined as

$$\varphi(x) = (x[0], x[1], f(x))$$

for some complicated function f. For example, maybe $f(x) = \exp(-\|x - (3,4)\|^2)$, a smooth bump centered around the landmark $(3,4)$, which domain knowledge says is salient.

← Food For Thought: With the same three landmarks, find a hyperplane in feature-space that produces these predictions:



(and we'll rely on gradient descent to select which of these are actually useful):

$$\varphi(x) = (\text{similarity}(x, x_0), \text{similarity}(x, x_1), \cdots \text{similarity}(x, x_{N-1})) \in \mathbb{R}^N$$

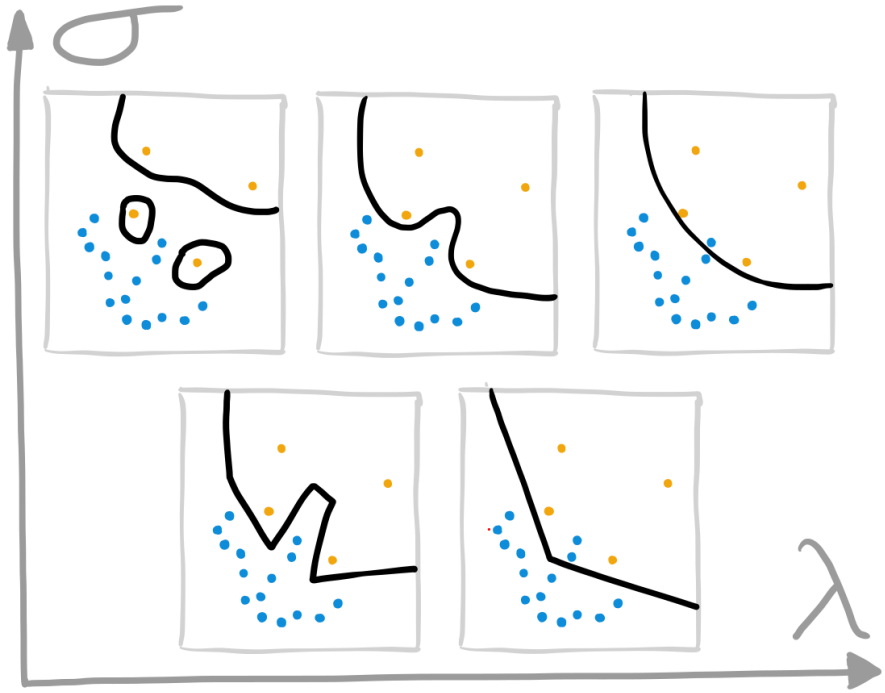
This is a *data-dependent* featurization: we adapt φ to training data.

At day's end, we'll use φ to make predictions the same way as before, namely via a hypothesis $(x \mapsto \text{sign}(w \cdot \varphi(x)))$. That is, we'll have a bunch of weights w_i and we'll classify a fresh x according to the sign of

$$\sum_i w_i \cdot (\text{ith feature of } x) = \sum_i w_i \cdot \text{similarity}(x, x_i)$$

With N features for N training points, $\mathcal{H}$ will (usually) be very expressive. We (potentially) pay for this improved approximation by suffering worse generalization: expressivity means $\mathcal{H}$ contains many hypotheses that well-fit the training data but do horribly at testing time. On the other hand, if we've done a good job choosing *especially informative* features, then $\mathcal{H}$ will contain a hypothesis that does well on both training and testing data. Thus, regularization is crucial!

Below we show how regularization affects learning when we use similarity features. Here we'll define similarity$(x, x') = \exp(-\|x - x'\|^2/\sigma^2)$; this popular choice is called the **Gaussian RBF kernel**. We minimize our familiar friend $\lambda\|w\|^2 + \sum_i \max(0, 1 - y_i w \cdot \varphi(x_i))$ for different settings of λ, σ.



Food For Thought: Is training data featurized with the $\sigma = 1$ Gaussian RBF kernel always linearly separable? That is, can we always achieve perfect training accuracy? Assume, of course, that no training points overlap. *Hint*: use huge ws.

Food For Thought: Design a similarity function to help recognize spam emails. For our purposes, an email is not just body text but also a timestamp, sender address, subject line, set of attachments, etc. TODO: DIMPLES EXERCISE

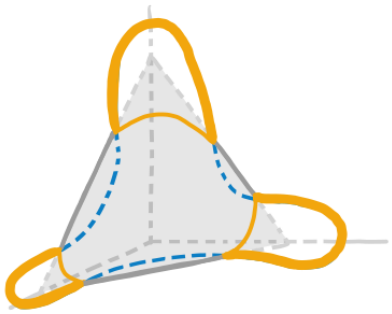


Figure 18: A hyperplane in feature-space that produces the predictions depicted in the margin figure on the previous page.

Figure 19: **The RBF kernel's σ controls spatial resolution.** As expected, λ controls model complexity. But so does σ! *Why?* Well, σ is a kind of 'blurring' parameter: the ith feature $\varphi_i(x) = \exp(-\|x - x_i\|^2/\sigma^2)$ is nearly constant once x gets closer than σ to the landmark x_i. *Intuitively, φ 'throws away' spatial information at resolutions $< \sigma$.* So for huge σ, the decision function will lack detailed wrinkles (unless w is huge). Conversely, when $\|x - x_i\|$ far exceeds σ, $\varphi_i(x)$ decays very rapidly. So for tiny σ, $\varphi(x)$'s entries will all be small — but in relative terms, the entry for the x_k closest to x will far exceed all other entries. So for tiny σ the hypothesis class easily expresses **nearest neighbors** rules of the form "predict whichever label the closest training point has."

SUPERPOSITIONS AND KERNELS — In this passage we'll discuss how, once we've featurized our xs by similarities, we'll select a hypothesis from $\mathcal{H}$ based on training data. As usual we can do ordinary gradient descent, the kind we're now used to. But we'll here explore a different method, a special gradient descent. The method is important because it offers a fast way to solve seemingly different problems:

ordinary (slow) gradient descent on data featurized as we please, say by $x \mapsto \varphi(x)$	*is equivalent to*	special (fast) gradient descent on data featurized according to similarity$(x, x') = \varphi(x) \cdot \varphi(x')$

Here's an analogy for the speedup. We're quick-sorting an array of objects-so-large-that copies are expensive. Instead of directly sorting, we create and sort an array of pointers to the original objects; only as a final step do we arrange the objects based on the sorted pointers. That way we do N large-object-swaps instead of N log N. Better yet, if the purpose of sorting was to allow us to quickly binary search to count the array elements x_k less than given xs, then we can avoid large-object-swaps *completely* (!) by binary searching the array of pointers.

Now for the two methods, ordinary and special. Both methods aim to reduce the loss $\ell(y_k, d_k)$ suffered at the kth training example (here we use the shorthand $d_i = w \cdot (\text{features of } x_i)$ for the decision function value at the ith training input). Whereas the ordinary method subtracts gradients of loss with respect to w, the special method subtracts gradients of loss with respect to d. We can write both in terms of the partial derivative $g_k = \partial \ell(y_k, d_k)/\partial d_k$:

ordinary, w-based update	special, d-based update
$w^{\text{new}} = w - \eta \dfrac{\partial\, \ell(y_k, d_k)}{\partial\, w}$	$w^{\text{new}} = w - \eta \dfrac{\partial\, \ell(y_k, d_k)}{\partial\, d}$
$= w - \eta\, g_k \cdot (\text{features of } x_k)$	$= w - \eta\, g_k \cdot (\text{kth one-hot vector})$

Note that w has as many entries as there are features and d has as many entries as there are training examples; so the special update only makes sense because we've chosen feature-set indexed by training points! Intuitively $d = X \cdot w$, so w and d are proportional and the ordinary and special updates are just stretched versions of each other. In multiple dimensions, different directions get stretched different amounts; it's because of this that using the two updates to classify similarity-featurized data gives different predictions at day's end.

But using the special update to classify similarity-featurized data gives the *same* predictions as using the ordinary update to classify data featurized as we please, say by $\varphi : \mathcal{X} \to \mathbb{R}^F$, so long as we define similarity$(x, x') = \varphi(x) \cdot \varphi(x')$ to be a dot product. In each ordinary update we do an F-dimensional dot product of weights with $\varphi(x_k)$. By contrast, in each special update we do an N-dimensional dot product of weights with x_k's similarity-features. We get a speedup when there are many more features F than training points N.

Food For Thought: Say $\mathcal{X} = \mathbb{R}$. We define a thousand features $\varphi_s(x) = \sqrt{x^s/s!}$ for $0 \le s < F = 1000$. Compute similarity(x, x) quickly, up to floating point error. Compute similarity(x', x'') by comparing $x = x', x = '', x = x' + x''$.

Food For Thought: Say $\mathcal{X} = \mathbb{R}^3$. Let's define similarity$(x, x') = \exp(-\|x - x'\|^2)$. Explain to a friend why 'similarity' is a fair name for that function. Find a $\varphi : \mathcal{X} \to \mathbb{R}^{1000}$ so that similarity$(x, x') = \varphi(x) \cdot \varphi(x')$ up to floating point error.

← This is probably the most challenging passage I'll write. I worked hard to explain kernels as simply as possible but no simpler. explain 'superposition!'

← Frankly, I don't know of modern projects where this 'kernel trick' speedup matters. With GPUs and big data and deep learning, the main computing bottlenecks lie elsewhere. Still, the kernel trick rewards study: it's an idea that continues to inspire modern techniques.

← Instead of using pointers to implicitly arrange an array of memory-hogging objects into an ordering that helps compute a rank for a fresh x, we'll use numbers to implicitly arrange a training set of high-dimensional featurevectors into a formal linear combination that helps compute a label for a fresh x.

← We use a fact that looks advanced but that's obvious once you get past the language:

$$\begin{aligned}\partial\, \ell(y_k, d_k)/\partial\, d &= g_k \cdot (\text{kth one-hot vector}) \\ &= (0, \cdots, 0, g_k, 0, \cdots, 0)\end{aligned}$$

This says that, since $\ell(y_k, d_k)$ depends on d only through the kth coordinate of d, the gradient will be zero in every direction except the kth one. It's a case of the chain rule. So *the special update only changes the* kth *weight entry!*

← Here, X is the N × N matrix whose kth row is the featurization of the kth training input. So $X_{ki} = \text{similarity}(x_k, x_i)$.

← Why? We'll see on the next page. For now we briefly discuss the upshot of this equivalence.

← Food For Thought: Say we have in RAM all N × F entries for the training φ-feature vectors. T ordinary updates would take FT multiplications; T special updates would take $FN^2 + NT$ multiplications. *Where does that* FN^2 *come from?*

← This similarity is called the **Gaussian RBF kernel**. We can also scale $\|x - x'\|$ (i.e., measure distance in different units) before exponentiating.

Each ordinary update adds some linear combination of training inputs to the weights, so (if we initialize weights to zero) we can after any number of steps write $w = \sum_i \alpha_i \varphi(x_i)$.° Then the decision function value for any x is

← those αs change as we change the weights

$$d = w \cdot \varphi(x) = \sum_i \alpha_i \varphi(x_i) \cdot \varphi(x) = \sum_i \alpha_i \text{similarity}(x_i, x)$$

We recognize this a sum of similarity-based features, weighted by αs! In other words, the F-dimensional weight vector $w = \sum_i \alpha_i \varphi(x_i)$ *makes the same predictions* on φ-featurized data as the N-dimensional weight vector $(\alpha_0, \cdots, \alpha_{N-1})$ makes on similarity-featurized data!

So our two scenarios have (effectively)° the same hypothesis class. To see why they moreover select from this class *the same* hypothesis, we compare the ordinary update for weight w against the special update for weight α:

← I say 'effectively' because the ordinary-with-φ scenario could have many more hypotheses corresponding to weight vectors not of the form $\sum_i \alpha_i \varphi(x_i)$. But gradient descent in that scenario wouldn't select those hypotheses anyway.

ordinary update on w	special update on α
$w^{\text{new}} = w - \eta\, g_k \cdot \varphi(x_k)$	$\alpha_k^{\text{new}} = \alpha_k - \eta\, g_k$

and then substitute $w = \sum_i \alpha_i \varphi(x_i)$. This whole idea is the **KERNEL TRICK**.

Okay, that's a lot to absorb. Let's see how it works for perceptron loss. Here $g_k = \partial \ell(y_k, d_k)/\partial d_k$ is $-y_k \cdot \text{step}(-d_k)$: zero if $d_k > 0$ and $-y_k$ otherwise. In

ordinary perceptron	kernelized perceptron
$w^{\text{new}} = w + \eta y_k \text{step}(-d_k) \cdot \varphi(x_k)$	$\alpha_k^{\text{new}} = \alpha_k + \eta y_k \text{step}(-d_k)$
$d_k = w \cdot \varphi(x_k)$	$d_k = \sum_i \alpha_i \text{similarity}(x_i, x_k)$

code, the kernelized perceptron looks like this:°

← We call `similarity_features(x[42])` again each time we update based on the $(k = 42)$nd example. This wastes time. It's faster instead to precompute the training examples' similarity features once and for all.

```
similarity_features = lambda x: np.array([similarity(x,x[k]) for k in range(N)])
predict = lambda alpha, x: np.sign(np.dot(alpha, similarity_features(x)))
alpha = np.zeros(N)
for k in IDXS: # sequence of training examples indices to update based on
  alpha[k] = alpha[k] + ETA * y[k] * (1 if predict(alpha, x[k]) != y[k] else 0)
```

The program never refers explicitly to φ; it only calls `similarity`.°

← We call such a program **kernelized**.

Food For Thought: Write a kernelized program to minimize hinge° loss.

← i.e., $\ell(y, d) = \max(0, 1 - yd)$, remember?

Food For Thought: That $w = \sum_i \alpha_i \varphi(x_i)$ suggests two different L2 regularizers: $\lambda\|w\|^2$ or $\lambda\|\alpha\|^2$. Write kernelized programs to minimize (hinge loss plus either regularizer). The $\lambda\|w\|^2$ is trickier.

We have flexibility in designing our function similarity : $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$. But for the function to be worthy of the name, it should at least satisfy these two rules: (**symmetry**) x is as similar to x' as x' is to x and (**positivity**) x is similar to itself.°

← In symbols, symmetry says $\text{similarity}(x, x') = \text{similarity}(x', x)$ and positivity says $\text{similarity}(x, x) \geq 0$.

Food For Thought: *Show that, if our similarity function disobeys the positivity rule, then a kernelized perceptron update on x_k can* worsen *the prediction for x_k!*

So the positivity rule helps learning. That rule generalizes to a yet more helpful one ("**Mercer's**"): if we expand our L2 regularizer $\lambda\|w\|^2 = \lambda\|\sum_i \alpha_i \varphi(x_i)\|^2$ into a weighted sum of similarities, that sum should never be negative.°

← You might say: the answer is $\lambda\|w\|^2$, so it can't be negative! That's true when we derive similarity from φ (for example, $\text{similarity}(x, x') = \exp(-\|x - x'\|^2)$ obeys Mercer). But my point is that we can invent similarity functions in other ways; those functions might or might not obey Mercer. Optimization works better when they obey Mercer.

Food For Thought: Say $\mathcal{X} = \mathbb{R}^2$. Does $\text{similarity}(x, x') = 1 + \|x - x'\|$ obey Mercer's rule? Our $N = 2$ training points are $\{(x_0, y_0 = +1), (x_1, y_1 = -1)\}$ with x_0 far from x_1. Manually work out one gradient step on the *total* perceptron loss. Sketch the decision boundary. Notice that it goes 'the wrong way'!

QUANTILES AND DECISION TREES — There are many other good ideas for choosing featurizations based on data. Here's one: *rescale a feature based on the distributions of its values in the training data*. VERY OPTIONAL

From quantiles to binning.

We won't discuss them in lecture, but **decision trees** can be very practical: at their best they offer fast learning, fast prediction, interpretable models, and robust generalization. Trees are discrete so we can't use plain gradient descent; instead, we train decision trees by greedily growing branches from a stump. We typically make predictions by averaging over ensembles — "forests" — of several decision trees each trained on the training data using different random seeds.

GENERALIZATION COST OF data-dependent featurization!

LINEAR DIMENSION-REDUCTION — There are many other good ideas for choosing featurizations based on data. Here's one: *if some raw features are (on the training data) highly correlated*, collapse them into a single feature. Beyond saving computation time, this can improve generalization by reducing the number of parameters to learn. We lose information in the collapse — the small deviations of those raw features from their average° — so to warrant this collapse we'd want justification from domain knowledge that those small deviations are mostly irrelevant noise.

← or more precisely, from a properly scaled average

More generally, we might want to

One way of understanding such linear dimension-reduction is matrix factorization. I mean that we want to approximate our $N \times D$ matrix X of raw features as $X \approx FC$, a product of an $N \times R$ matrix F of processed features with an $R \times D$ matrix C that defines each processed feature as a combination of the raw features.

There's **principal component analysis**.

As a fun application, we can fix a corrupted row (i.e., vector of raw features for some data point) of X by replacing it with the corresponding row of FC. We expect this to help when the character of the corruption fits our notion of "$\approx$". For example, if the corruption is small in an L2 sense then PCA is appropriate.

collaborative filtering

Learning Features from Data

shallow learning

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

- train (and make predictions using) a shallow binary classifier
- visualize learned features and decision boundaries for shallow nets
- derive and intuitively interpret shallow nets' learning gradients

Thanks to lpamarescot for help!

So many learners volunteered to help write notes. Alas, it turns out there are weird legal barriers to sharing our LaTeX source code, so my original idea of sharing latex source that y'all could improve will not happen. Another challenge has been more on me: I've been running behind on drafting notes to begin with!

A pale but still-meaningul substitute: *if you have any suggestions for changes to the notes, please tell me!*

A MENU OF FEATURIZATION FUNCTIONS — We've discussed linear models in depth. We've seen how important it is to prepare the data for linear models by choosing appropriate featurizations — for example, applying the $(x \mapsto (1, x))$ bias trick can drastically improve testing accuracy! So we've *hand-coded* non-linearities to extract usable features from raw features. This makes our models more expressive.

Now we'll discuss how to *learn* features from data.° This idea is called 'deep learning'. The word 'deep' references that soon we will layer feature-learners on top of each other. But we'll start simple, with just one feature-learning 'layer'.

← We came close to this when talking about kernel methods. Kernel methods use a featurization that depends on the training inputs. But, intuitively, that featurization isn't particularly 'fitted' (e.g. we haven't chosen our kernels by gradient descent on some interesting loss). Less importantly, the featurizations we used when using kernels don't depend on the training outputs.

Let's build a logistic classifier that *learns* the features it ultimately separates-via-hyperplane. In brief, the classifier will be described by a weight matrix A that combines those features just as in Unit 1, together with a weight matrix B that *defines* those features in terms of the raw input. The numeric entries of A, B change during training; as B changes, the features it defines change to become more useful, and this is what we mean when we say that we "learn features". We'll call this classifier a **shallow neural network**. Note, however, that those two matrices' numbers of columns and of rows stay the same during training; we specify those shapes as part of our design process. So, while we no longer handcraft features (i.e., do manual feature selection), we still choose how many features to use and what the "allowed shapes" for each feature are. Those choices are part of **architecture**. We can tune architectural choices as we do other hyperparameters, for instance by cross-validation.

ADD image

Figure 20: A toy example of the decision boundary (**black**) of a shallow neural network on 2D inputs (preprocessed with the bias trick). This neural network has 8 features (shown as subtle discontinuities in shading, with less shading when that feature is negative and more shading when that feature is positive), and we depict the weight on each feature by the shading's saturation. The next couple pages explain how we build a model that learns such features and can have such decision boundaries.

The above is our roadmap. Let's see how B actually appears in our math. We want the classifier to learn a featurization function that maps each an input x, represented via raw (or 'rawer') features, to some representation $\tilde{x}$ more useful to the task. That is, we will present to the machine a menu $\{\cdots, \varphi, \cdots\}$ of possible featurization functions and we want the machine to select a particular function φ to use as a way to translate raw features x to features $\tilde{x}$.

What menu should we use? Well, we've already seen (hardware and theory) advantages in defining a menu by giving a function φ_B for each matrix B, where $\varphi_B(x)$ somehow relates to the product Bx. But φs must be non-linear in order to increase expressivity. So let's process Bx through a nonlinear function f:

$$\tilde{x} = f(Bx)$$

We define our menu of possible featurizing functions as the set of functions of

the above form. The whole menu shares the same f; menu items differ in their Bs.

What f shall we use? Commonly used fs include the ReLU function $\text{relu}(z) = \max(0, z)$ and variants.[◦] But keep in mind that we often encounter situations where domain-specific knowledge suggests special fs other than ReLU variants. Anyway, let's use the "leaky" ReLU variant $\text{lrelu}(z) = \max(z/10, z)$. Actually, for z an array we will do that operation on each component separately, and we'll throw in the bias trick:

$$f(z[0], z[1], \cdots) = (1, \max(z[0]/10, z[0]), \max(z[1]/10, z[1]), \cdots)$$

So overall our logistic classifier represents the probability model:

$$\hat{p}(y{=}{+}1 \,|\, x) \;=\; \sigma(A(f(Bx)))$$

We have a hypothesis for each (A, B) pair. Here A is our familiar weight vector that linearly separates features $\tilde{x}$; what's new is that a B influences how $\tilde{x}$ depends on x. We want to learn both A and B from data.

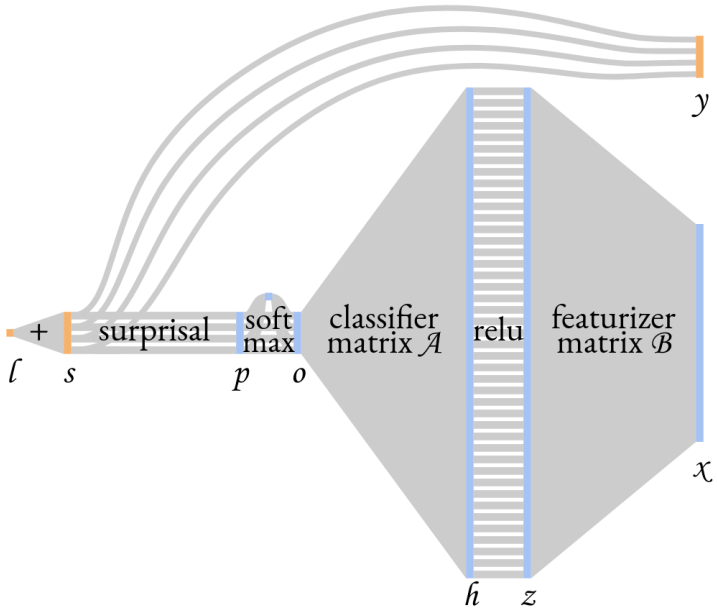


Figure 21: **Architecture of shallow neural nets.** Data flows right to left via gray transforms. We use the blue quantities to predict; the orange, to train. Thin vertical strips depict vectors; small squares, scalars. We train the net to maximize data likelihood per its softmax predictions:

$$\ell = \textstyle\sum_k s_k \quad s_k = y_k \log(1/p_k)$$

$$p_k = \exp(o_k)/\textstyle\sum_{\tilde{k}} \exp(o_{\tilde{k}})$$

The decision function o is a linear combination of features h nonlinearly transformed from x:

$$o_k = \textstyle\sum_j A_{kj} h_j$$

Each "**hidden activation**" or "**learned feature**" h_j measures tresspass past a linear boundary determined by a vector B_j:

$$h_j = \text{lrelu}(z_j) = \max(z_j/10, z_j) \quad z_j = \textstyle\sum_i B_{ji} x_i$$

We've not depicted a bias term but you should imagine it present.

Let's recap while paying attention to array sizes. Our logistic classifier is:

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

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:

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

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).

← Older projects often use functions like $f(x) = \tanh(x)$. We'll see such functions show up playing specialized roles (e.g. in LSTMs, which themselves are now a bit outdated for their main use case in language). But experience has now shown that these functions aren't a good default choice.

ACTIVATION FUNCTIONS — Here's a brief aside on why we use activation functions such as lrelu. *I don't want to overemphasize this topic*, even though it is important, since it's best appreciated once you've gotten your hands dirty with code, and we don't want to miss the forest for the trees on our journey to that code-writing stage.

Food For Thought: Sketch $\text{lrelu}(z), \text{relu}(z), \tanh(z)$ against z. See $\text{lrelu}(z)$'s two linear pieces with different slopes; its derivative is never very close to 0 and this eliminates one of the major ways that gradient descent can get "stuck", namely, (to use a physics analogy) the way a ball can get stuck atop a mesa's flat plateau instead of falling down the cliff. This is the so-called **vanishing gradient** problem. *This problem does not affect our Unit 1 linear models. Why is this?*

Food For Thought: Here's a toy example of vanishing gradients. Let's use a model $f_{a,b,c}(x) = a + b\tanh(c + x)$, with a, b, c, x all numbers.° We initialize $(a, b, c) = (0, 0)$ and run gradient descent (GD) with least-squares loss on three datapoints $(x, y) = (-60, -1), (-40, -1), (-20, +1)$. This data is well-explained by $\theta_\star = (a, b, c) = (0, 1, 30)$. But the weights take a very long time to get near $\theta_\star$. Do you see why? What if we use relu instead of tanh? *What if we use lrelu?*

← You'll recognize a, b as analogous to the matrix A in our above architecture and c as analogous to the matrix B.

Especially before ~2018, and especially in deep, "dynamic" models such as RNNs and GANs, the vanishing gradient problem was severe. Nowadays we use techniques such as "batch normalization", "adaptive gradients", and lrelu to cure the vanishing gradients problem.

TRAINING BY GRADIENT DESCENT — Gradient descent works the same:

$$w \leftarrow w - \eta \nabla \ell(w)$$

where $w = (A, B)$ consists of all learned parameters (here, the coeffiecients of both A and B), ℓ is the loss on a training batch, and η is the learning rate.

We've already learned how to compute $d\ell/do$ and $d\ell/dA = (h)(d\ell/do)^T$ (to use the notation of the architecture figure). Likewise we may compute $d\ell/dh = A^T(d\ell/do)$. To address the nonlinearity we use the chain rule:

$$d\ell/dz_k = (d\ell/dh_k) \cdot \text{lrelu}'(z_k) \qquad \text{lrelu}'(z_k) = (1/10) \text{ if } z_k < 0 \text{ else } 1$$

and finally, in strict analogy to $d\ell/dA = (h)(d\ell/do)^T$, our B-gradient $d\ell/dB = (x)(d\ell/dz)$. This process of working backward using the product rule and chain rule is called **backpropagation** — it's an organized system for computing derivatives efficiently.

Intuitively, $d\ell/dA$ tells us how to re-weigh the features we have while $d\ell/dB$ tells us how to change our features. The image I have in mind is of shifting pressure between one's legs vs sliding one's feet across the floor.

One more thing — **To break symmetry, we should initialize with (small) random weights rather than at zero. Do you see why?**

Let's see what these gradient dynamics look like. Our decision boundaries look more complicated, as expected. We also depict each learned feature.

Next 3 questions: we initialize $A = B = 0$ and work qualitatively/roughly.

Food For Thought: What is the training loss at initialization?

ADD images

Figure 22: **Training dynamics of a shallow neural net**. We use artificial 2D data. The net has 8 hidden features, which we depict as subtle edges between colors. In bold are overall decision boundaries. In English reading order (left-to-right top row, then left-to-right bottom row), we show the network after 0, 500, 1000, etc many gradient steps. **Notice the features 'swinging around' to better capture the patterns in the data.**

Food For Thought: What is the loss gradient at initialization?

Food For Thought: What is the testing accuracy after a thousand SGD updates?

For the questions below, assume a fixed, smallish training set and a fixed, moderate number of gradient descent steps. Work qualitatively/roughly.

Food For Thought: what should the training and testing accuracies look like as a function of hidden dimension?

Food For Thought: what should the training and testing accuracies look like as a function of the learning rate?

REGULARIZATION — Here is a simple way to generalize L2 regularization for shallow neural networks:

$$\text{regularization penalty} = \lambda_A \|A\|^2 + \lambda_B \|B\|^2$$

Here, $\|A\|^2, \|B\|^2$ are the sums of the squares of the entries of those matrices. This leads to the gradient terms

$$A^{\text{new}} = A - \eta \cdot (\cdots + 2\lambda_A A) \qquad B^{\text{new}} = B - \eta \cdot (\cdots + 2\lambda_B B)$$

Here, the $\cdots$ represent the gradients from the non-regularization terms.

It is often a good idea to regularize bias coefficients by a different amount — potentially by 0 — than the λs used for the other weights.

Food For Thought: Observe that L2 regularization disambiguates the scaling redundancy coming from our choice to use lrelu activations. What I mean by "scaling redundancy" is that if we change (A, B) into $(A', B') = (A/68, 68B)$, then we get the same predictions: $A \cdot \text{lrelu}(B \cdot x) = A' \cdot \text{lrelu}(B' \cdot x)$.

The $\|A\|^2$ term favors *large margins with respect to the learned features*, just as we saw for linear models. When the $\|B\|^2$ term is small, large margins with respect to the learned features imply *large margins with respect to the raw inputs*. The two terms work together; if we just used one of them, the aforementioned scaling redundancy would allow the network to have small margins with respect to raw inputs.

As we move toward deep learning, we will start focusing on "implicit" or "architectural" methods of regularization as supplements to and even replacements for the explicit regularization as above. You can also google "batch normalization" or (for historical interest but a bit outdated) "dropout". Google stuff and then ask questions in the forum; I'll try to answer.

A TOY EXAMPLE — Here's a toy dataset that can help develop a mental model for shallow neural networks. It's binary classification of 3-D input vectors within the cube $[-1,+1] \times [-1,+1] \times [-1,+1]$. We'll write the components of each vector as (x_0, x_1, x_2). The points all obey $x_2 = \epsilon \cdot \text{sign}(\min(x_0, x_1))$ for $\epsilon = 1/10$. The points are classified as positive or negative according to whether their x_2 is positive or negative. The data is uniformly distributed across the described region.

So this dataset is linearly separable. Nevertheless, we want to classify it using a shallow neural network with 2 hidden features. For simplicity, we use hinge

loss on the values $Af(Bx)$ instead of logistic loss. Also for simplicity, we do the bias trick neither on the raw inputs nor on the learned features. So A has shape 1×2 and B has shape 2×3. We constrain A and the two rows of B to be unit vectors; this models the effect of regularization by keeping all weights small-ish. Food For Thought: The shallow neural net can mimic a linear classifier by setting $A = [-1, 0]$ and $B = [[0, 0, -1], [1, 0, 0]]$. *In which direction will gradient descent tilt* A *and* B*?* And, after taking a few gradient steps, *how does the shallow neural network's decision boundary change from a linear hyperplane?* Think qualitatively, not quantitatively.

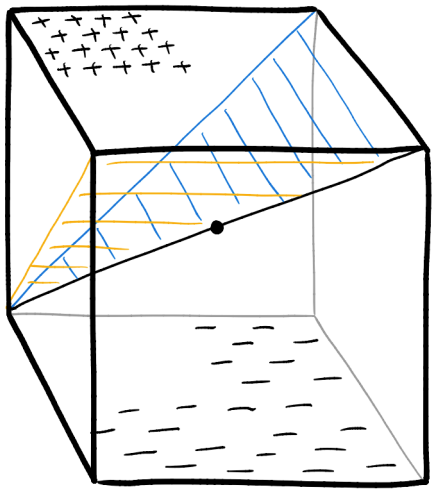


Figure 23: An illustration of the training data (black $+$s and $-$s) and of a "butterfly" shaped decision boundary that the shallow neural network can express. We defined the training data with $\epsilon = 1/10$ ($+$s and $-$s closer together vertically) but here we depict the training data with $\epsilon = 1$ to make the figure easier to understand.

TOWARD DEEP NEURAL NETWORKS — Inspired by the idea of turning raw inputs into more useful features, we can iterate, turning those learned features into even more useful learned features:

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

Here, the matrix C turns x into a feature vector of dimension $\tilde{h}$, then the matrix B turns *that* into a feature vector of dimension h, then the matrix A classifies based on those super-duper learned features. In jargon, we say that this model has "2 hidden activation layers" (counting those two fs) or "3 weight layers" (couting to those 3 matrices). And we can keep going. Models with dozens of layers are now the norm; models with hundreds of layers have been successfully explored. More jargon: the number of layers is **depth**; the dimension of each layer is **width**.°

← Once you understand how to derive gradient descent for the shallow case, deriving gradient descent for deeper networks will feel easy.

Both depth and width increase our model's complexity. They do so in different ways. Insofar as specific values for weight matrices define a "program" that transforms inputs to outputs, large *depth* allows the definition of helper functions in terms of primitives while large *width* allows the direct combination of many primitives.

By analogy, we might say that a lecturer has "expressive grammar" if they combine words into phrases, phrases into clauses, clauses into sentences, sentences into paragraphs, in intricate combinatorial patterns full of nuance. We might say a lecturer has "expressive vocabulary" if they use just-right, not-so-common words to vividly capture their meaning. Depth allows complex grammar. Width allows complex vocabulary.

For example, imagine a NN for processing images of what we have in our fridge. Its first layer features can detecting basic color patterns (such as streaks of red, leafy edges, shadows of something bulky, etc) but not for detecting "higher" concepts such as *red apple* vs *red cherry*.

The second layer starts with these basic color patterns as inputs, so it can build in complexity. It can detect fruit parts by how those basic color patterns "hang together": if a small leafy edge appears directly above a small shadowy region, then it is more likely to be a true 3d leaf rather than a 2d sticker that happens to be green; if a small red streak appears next to a bright white segment, then it is more likely to be part of a beef-atop-styrofoam package than a patch of ripe apple.

The layer after that takes as input these fruit-part measurements — one of

those inputs will be high when seems to be a leaf near the image's center; another of those will be high when there seems to be a patch of apple near the image's left. By combining these inputs, it can detect actual fruits and distinguish far-away red apples from nearby cherries, even though both look like red disks of the same sizes.°

Neural nets are good at squeezing as much as they can out of correlations in the training data.° So, if we are training an apple-vs-cherry classifier, and if in our training examples apples tend slightly to co-occur with crammed fridges° and cherries tend slightly to co-occur with sparse fridges, then the neural net will pick up on this. It will learn not just what local color patterns look more like cherries or apples but also what global color patterns look more like crammed fridges vs sparse ones. Sometimes the network's learning of such a feature is desirable; other times, not.

But this packet of notes is supposed to just lay the groundwork for Unit 3. I'll try to discuss more about depth, CNNs, and RNNs in future packets of notes!

← By the way, the story in these paragraph is optimistic. It is true as long as we are okay with mediocre (but still much better than chance) accuracies. In practice, unless the image data is especially simple, one should try more layers for computer vision.

← In this whole section, I'm making general claims about usual architectures trained by usual methods. I am not claiming universal laws.

← Perhaps because the kind of fridge-user who refrigerates apples tends to refrigerate all of their food. In the cold climate of Michigan, where I'm from, we usually don't have to refrigerate apples.

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
```

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:

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

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" —

deep learning: ideas in architecture

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

-
-
-

LAMARCKISM, LOCALITY, DEPENDENCE — The key theme of deep learning is what I'll cheekily call the Lamarckian motto: **To build a tool, use it!**. It's as if hitting a flabby, asymmetrical hammer against a lot of nails gradually calloused the hammer to be exactly the right hardness and shape to drive the nails straight and deep, or as if we could lift ourselves by our bootstraps. For example, in last section's shallow

neural network we wanted to build a featurization useful for linear classification. To *build* that featurization, we *used* it: we piped a differentiably parameterized featurizing layer's output into a linear classifier and we trained both the featurizer and the classifier using gradient descent.

We can summarize the situation by drawing learned functions as dashed arrows and hardcoded functions as solid arrows (see margin). Thus, we have a natural distribution over *labeled images*, from which one function maps to labels and another maps to images (two solid arrows). Our shallow neural network consists of a featurizer and a classifier (two dashed arrows). We can get from a labeled image to a label by two paths: (a) directly, or (b) by mapping to just the image, then featurizing, then classifying. The condition we use to frame our machine learning problem — and the objective of gradient descent — is that the function (a) and the composition-of-functions (b) should be nearly equal when evaluated on naturally occuring labeled images.

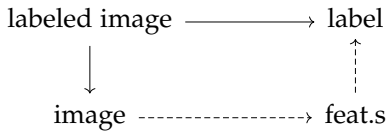
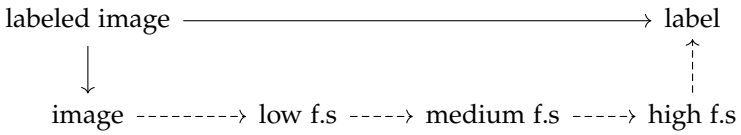


Figure 24: Architecture diagram of shallow neural classifier. There are two ways to get from the upper left to the lower right corner; we want them to agree.

That's all a complicated way of saying that we can improve raw features into classifier-friendlier features by gradient descent. Gradient descent on what? Not on the raw-to-friendly function in isolation, but on its composition with a classifier. We can evaluate how well the composition fits the training data but not how well the raw-to-friendly function by itself fits the data.° This is one of multiple connotations of the jargon adjective **end-to-end** when applied to architecture.

← There aren't any paths in the diagram that, like the diagram's bottom edge, go from images to features! So there's no path to directly compare that function to. This talk about paths is a complicated way of saying something that here is obvious. But path language can clarify more complicated situations.

So we can improve raw features into classifier-friendlier features. If something is fun, let's try it more! Let's do multiple layers of improvement. That is, instead of learning to turn raw inputs into features that are easy to classify, let's learn to turn raw inputs into "low-level features" that are easy to turn into "high-level features" that in turn are easy to classify. We can keep going. Here is how the idea looks if we have low, medium, and high features (as before, dashed arrows depict learned functions):



Concretely, this binary classifier could have a formula such as:°

$$\hat{p}(y{=}{+}1\,|\,x) \;=\; (\sigma^1_1 \cdot A^1_{h+1} \cdot f^{h+1}_h \cdot B^h_{\tilde{h}+1} \cdot f^{\tilde{h}+1}_{\tilde{h}} \cdot C^{\tilde{h}}_{\hat{h}+1} \cdot f^{\hat{h}+1}_{\hat{h}} \cdot D^{\hat{h}}_d)(x)$$

← Here, σ is the logistic function, f is an nonlinearity such as componentwise-leaky-ReLU-followed-by-bias-trick, and A, B, C, D are matrices. We write $\cdot$ instead of $\cdot$ for function composition, to reduce clutter. We annotate functions and matrices by a superscript dimension of output and a subscript dimension of input.

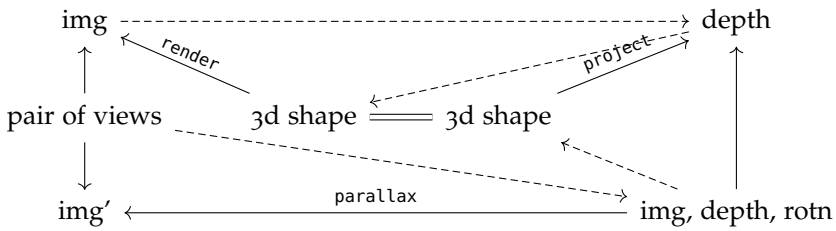
Here, $f(D(x))$ gives the low-level features, $f(C(f(D(x))))$ gives the medium-level features, and so on. The matrix A does familiar Unit-1-style linear classification of the super-duper learned features $f(B(f(C(f(D(x))))))$. Intuitively, a single featurization layer such as $f \cdot D$ can't do much by itself (at least, for computationally practical hidden dimensions and training cycles), so sequencing such layers helps simple transforms to accumulate into complex transforms. It's a bit like origami — simple folds build on simple folds to give a beautiful crane. By increasing depth, we give gradient descent a deeper hierarchy of folds to play around with as it tries to invent a crane.

I'm unhappy with the lack of categorical product below for the collection of features as inputs to 'high'

in subsequent diagrams, single arrows may have many layers

On with examples of the Lamarckian motto of building parts by exercising them in context. Let's say we want to learn how to turn an image of a natural scene (say, a photo of some furniture) into a depth map°. As is often the case in ML, we have huge amounts of data *but not huge amounts of data directly and reliably labeled for our task of interest*. Perhaps instead we have just plain images, or maybe, slightly better, pairs of slightly different shots of the same scenes. We want to be clever about squeezing learning signals from this data. From an individual image we might not always be able to specify the depth map uniquely, but at least we should have a consistency condition: there ought to exist some 3d mesh shape that induces the depth map and that renders to the given image (this rules out depth maps that are television-noise or M.C.Escherian, but it imposes constraints on our learned functions beyond this). So let's *learn* a depth-to-3dshape function so that we may express this consistency condition. It's okay if the learned function ends up being imperfect; it's a helper function to refine the learning of our depth map. And a pair of slightly different shots of the same scene should in principle usually have enough information to determine the depth map for both shots, as well as determining a small rotation to turn one shot into the other shot. So we have two kinds of consistency condition:

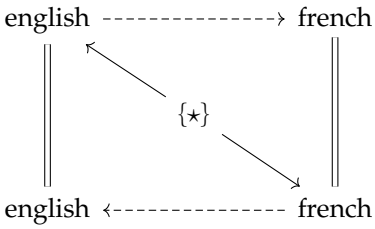
← i.e., an array that says for each pixel in the original photo how many meters away from the camera the surface depicted by that pixel is; for convenience we'll also regard the depth map as containing the data of the original image, too



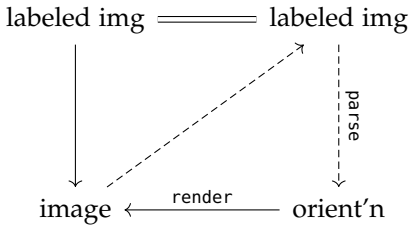
Zooming out, this key theme of deep learning suggests that we can build sophisticated systems by imposing consistency conditions between their parts.

Here are six further examples of the key theme:

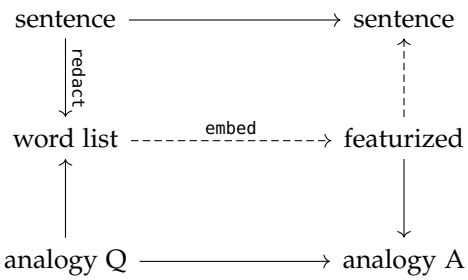
UNPAIRED TRANSLATION



ROBUST CLASSIFICATION



WORD EMBEDDING



CONCEPT HINTS

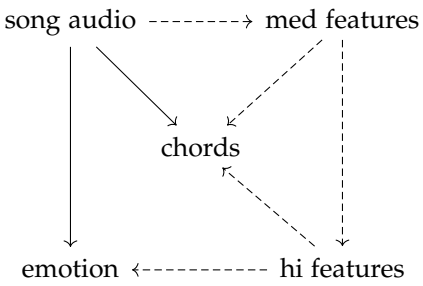
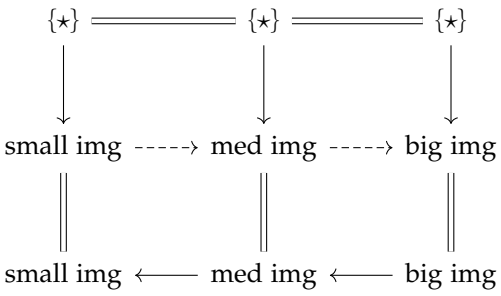
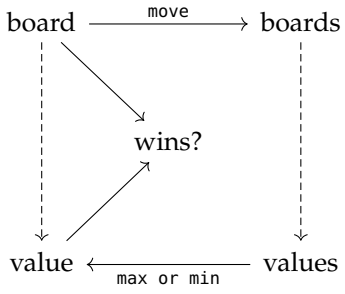


IMAGE GENERATION



BOARD GAMEPLAY



SYMMETRY, CONVOLUTION, ATTENTION —

HIERARCHY, REPRESENTATIONS, TRANSFER —

UNCERTAINTY, STRUCTURE, GENERATION —

BACKPROPAGATION AND GRADIENT DESCENT —

VISUALIZING LEARNING DYNAMICS —

VISUALIZING LEARNED REPRESENATIONS —

EXAMPLE CODE —

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

-
-
-

symmetry and locality: convolution

2. multiple layers

We can continue alternating learned linearities with fixed nonlinearities:

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

FEATURE HIERARCHIES —

BOTTLENECKING —

HIGHWAYS —

3. architecture and wishful thinking

REPRESENTATION LEARNING —

4. architecture and symmetry

symmetry and locality: attention

5. stochastic gradient descent

6. loss landscape shape

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

Modeling Structured Uncertainties

probabilistic models

Probabilistic Models —

[[[Data comes from somewhere]]] In this unit we explore the idea that *data comes from somewhere*. The data that we typically care about arose through processes we can to good approximation say a great deal about; then by a kind of reverse engineering of this process we can design good inference algorithms for this data. Probability theory often gives a good language for describing these processes: some random occurrence happened and based on that some other random occurrence happened and so forth until our data gets measured, so that overall we get this very complicated probability distribution whose richness we want to model. This fact — that data comes from somewhere — and this probabilistic attitude toward this fact will inspire us toward even richer and more sophisticated and more domain tailored architectures for our machine learning models.

Indeed, we started in Unit 1 with linear models. In Units 2 and 3 we enriched those models by pre-processing our input data (the input of our linear map) through sophisticated, domain-tailored non-linear featurizers. Now, in Unit 4, we will post-process the decision function value (the output of our linear map) through sophisticated, domain-tailored probability-related nonlinearities.

We've already seen two very small examples of this post-processing. First, softmax. Softmax takes a vector (the one that our linear map outputs) and squashes it into a probability distribution over a fixed finite set of possibilities. Then minimizing logistic loss is just maximum likelihood for this family of conditional probability distributions. Second, least squares regression. When we do regression, we get some numeric output for each input. We usually don't expect the output to be exactly correct; instead, we interpret the model as predicting that the truth will be close to its output. To be definite, let's regard the model as stating that the truth follows some fixed variance Gaussian centered on the output. Then minimizing square loss of a regression model is the same as maximizing likelihood, if we interpret the linear map's output as parameterizing the mean of a fixed-variance Gaussian.

In either case, the linear map's output picks out a particular distribution over the set $\mathcal{Y}$ of distributions, out of a family of such distributions. the post-processing we mention is simply the implementation of this picking out — jargon: *parameterization* — step. So our overall architecture will look like this: we take the input x in $\mathcal{X}$; then we feed it through some nonlinearity; then we apply our learned linear layer; then the output of that layer we regard as the parameters of some probability distribution over $\mathcal{Y}$.

That probability distribution exists only mathematically in the sense that in RAM it's just represented as its parameters. Still, we can usually do things like a sample from the distribution to get a diversity of concrete ys, or figure out the chance of any given y relative to the distribution. So our architecture overall

represents a family (parameterized by weights θ) of conditional distributions $p_{y|x;\theta}$ — or, what's the same, a family of stochastic functions from $\mathcal{X}$ to $\mathcal{Y}$.

An aspect of this enrichment is that the learning problem look quite interesting even in the very simple case where there's only one possible input x in $\mathcal{X}$. That is, when $\mathcal{X}$ has size one. Then our examples are just a bunch of ys and we seek to fit a distribution to that bunch. This is so-called *unsupervised learning*. Supervised learning and unsupervised learning are both instances of learning from examples. *Supervised learning* is the jargon that we use when we are especially interested in the challenge of learning input output relationships. *Unsupervised learning* is the jargon we use when what's more interesting to us is modeling the rich structure of the outputs. The distinction is as blurry and as practically useful at the distinction between a hotdog and a sandwich.

For most of this unit, we will focus on unsupervised learning, since the probability stuff is the new stuff and the input-output stuff we've already seen. We will occasionally sprinkle in examples with nontrivial input-output relationships, to show how that would work. Only near the end of this unit will we do this in depth.

[[[example: beyond gaussian for regression]]] Our architectures so far model uncertainty crudely. For example, when we train a neural network regression model f with weights θ and square loss, we are maximizing the following likelihood function (with independence assumption on training data):

$$p_{y|x}(y|x;\theta) = \text{gaussian distribution with mean } f(x;\theta) \text{ and variance } 1$$

In many real-world scenarios, however, the "true" distribution of y conditioned on $x = x$ is much richer than a gaussian distribution: it might have multiple humps, heavier tails, or correlations with other predictands. Think, for instance, of predicting the a person's distribution of daily-app-usages based on their age and job and country (but not on the day): for a musician maybe some days have a lot of tuning and sheet music usage and other days (vacations) have none, but with no days having intermediate levels. This is non-gaussian.

When the space Y of possible outputs is high-dimensional (e.g., a space of all images), there are even more ways for the true likelihood functions' shapes shape to differ from a gaussian.

FILLIN images

The distribution of all natural grayscale images is not gaussian when we represent images in "raw" pixel-brightness coordinates. I mean that we are thinking about multi-variate gaussians with as many dimensions as there are pixels. Natural images are all and only those photographs taken by professional photographers in the past four decades, made into grayscale, and cropped to a standard shape.

FOODFORTHOUGHT: What would a gaussian distribution in image space look like with spherical covariance? Arbitrary covariance? (Don't worry much about overflow beyond the maximum and minimum brightnesses).

What are qualitative aspects of the distribution of natural images that this does

not model? In a scenario where the distribution over natural images is gaussian, then the pixelwise average of any two images would be more probable than at least one of the images. But if (in aforementioned coordinates) we average a cat image and a dog image, we get an unclear image (rather than a clear image of an imaginary hybrid mammal). This means MULTIPLE MODES. PICTURE OF GAUSSIAN PROPERTY (This rhymes with non-convexity of the log probability landscape)

[[[forward picture of distribution (generative model)]]]

So in this unit, we want to treat the construction of richer, more domain-appropriate probability models as a design challenge, a degree of freedom we have as engineers. We'll go beyond the defaults such as square loss implicit in previous units. In doing so, we will engage with new facets of approximation. And the language we develop will clarify for us aspects of generalization that pertain to this as well as previous units, which will suggest to us a new and harder optimization goal. So we will then develop new ideas in optimization toward that goal.

That's abstract. The models we'll think about concretely in this unit are combinations of deterministic transforms and sampling statements. For example, the following is a model of length-N lists $[y_0, \cdots, y_{N-1}]$ of numbers. FOOD-

given is a number c_a
given is a number c_b
for $0 \leq n < N$:
    sample f_n by a coin flip that lands a with chance 0.8 else b
    sample y_n by normal distribution with mean c_{f_n} and variance 1

FORTHOUGHT: For $c_a = -5$ and $c_b = +5$, what might a 1D scatterplot of y_ns look like?

To model (x, y) pairs we might do something like:

given is a weight vector θ_a given is a weight vector θ_b
for $0 \leq n < N$:
    sample f_n by a coin flip that lands a with chance 0.8 else b
    sample y_n by normal distribution with mean $\theta_{f_n} \cdot x_n$ and variance 1

By contrast, the models we have looked at in previous units don't really do sampling except of y at the end (and maybe of θ at the beginning, for the prior).

TODO: compare (discuss conditional, compare to supervised; initial samplers priors and regularizer?)

[[[latents and marginalization]]]

[[[point estimate vs distribution; utilities; square loss muddies]]]

Example: 3State Traffic. Normalization. —

[[[3state traffic model definition]]] Now we describe a toy example we'll use throughout this Unit. There are 3 possible weathers each day in each city: Sunny, Hailing, and Foggy (SHF). We choose a city whose coldness ranges thru $[0, 1]$ (Warm thru Cold). And there are 2 possible traffics: NoJam or Jam. In a com-

pletely Warm city or a completely Cool city, the probabilities of weather-traffic pairs are proportional to: Cities with intermediate coldnesses linearly interpolate

warm:	Noj	Jam
S	.5	0
H	0	.25
F	0	.25

cold:	Noj	Jam
S	.05	.05
H	.4	.05
F	.4	.05

between these two tables.

The idea is that warm cities more often have sunny weather than cold cities. And that sunny weather tends to lead to less traffic jams — but in cold cities, where folks are used to non-sunny weather, so the effect of weather on jams is less.

We observe the traffic variable on a single day (one bit of information), and we want to estimate the city coldness (one real number in $[0, 1]$). This toy is very simple — misleadingly simple in some ways. We can solve this model exactly by thinking about it. Still, we'll soon use it to illustrate heavy-duty machine learning data crunching techniques. That way we can compare with the exact right answers.

FOODFORTHOUGHT: how would you compute the maximum likelihood estimate for coldness given Nojam? Given Jam? That is, which numbers above are involved in the arithmetic, and how are they involved in the arithmetic? Which case (Jam or NoJam) leads to a higher Coldness estimate?

[[[Sampling and Graphical Notation]]] [[[Inference with Known Parameter]]] [[[Parameter Estimation]]] [[[Challenge of Normalization]]]

Example: Gaussian Mixture Models. D-separation. —

[[[1-cluster case: forward model, generation, and inference]]] [[[enriching 1 cluster to 2 clusters. GMM definition. use cases.]]] [[[graphical model. question of fitting. explaining away.]]] [[[sampling from a 3-cluster GMM to make color / stick figure data]]] [[[example of conditional 1-cluster, 2-cluster GMM to connect to unit 3. color data]]]

Example: hidden markov models. Hierarchy and Transfer Learning. —

[[[definition of HMM]]] Here is an example of a probabilistic model for generating length-$T+1$ sequences of labels from a finite set L of label values.

given is a distribution $\iota(\cdot)$ on L
given is a conditional distribution $\tau(\cdot|\cdot)$ from L to L
sample l_0 from $\iota(\cdot)$
for $0 \leq t < T$:
    sample l_{t+1} from $\tau(\cdot|l_t)$

MARKOV PROPERTY

[[[]]] Adding a layer of sophistication, we might imagine that our observation at timestep t is related to but not the same as the evolving state.

FAILURE OF MARKOV PROPERTY

given is a distribution $\iota(\cdot)$ on Z
given is a conditional distribution $\tau(\cdot|\cdot)$ from Z to Z
given is a conditional distribution $\epsilon(\cdot|\cdot)$ from Z to L
sample z_0 from $\iota(\cdot)$
for $0 \leq t < T$:
    sample z_{t+1} from $\tau(\cdot|z_t)$
for $0 \leq t \leq T$:
    sample l_t from $\epsilon(\cdot|z_t)$

[[[Expressivity: Generating Cool Sequences]]] [[[Inference given Weights: os and Softlogic and Backflow]]] [[[Inference of Weights, Qualitatively. Hierarchy and Transfer.]]]

Latents, Marginal Likelihood for Latents, Weights as Latents. —

[[[Weights as Latents.]]] [[[Point vs Distribution estimates, and generalization.]]] [[[Bowtie vs Pipe]]] [[[BIC as an approximation. Double descent.]]] [[[Beyond BIC]]]

inference by variation: EM

EM overview —

[[[Challenge of summing]]]

The idea of expectation maximization is to do coordinate ascent there are two things we don't know there's theater which is our private or we want to find a single best value for and there is Z of the latent random variable whose multiplicity of possible values will you wish to account for we have to treat theater and see slightly asymmetrically because what we really want to find its data that maximizes a certain some over zs

[[[EM qualitatively: maintain many replicas with different zs]]]

[[[E and M steps: formulas]]]

mention gradient descent as option in M mention how to incorporate priors

[[[cartoon of EM for GMM]]]

[[[cartoon of EM for HMM]]]

EM: GMM example (more detail in pset) —

[[[type signatures of E step and of M step]]] [[[qualitative behavior of E step and of M step]]] [[[formula for M step]]] [[[formula for E step]]] [[[example dynamics of EM on flower data]]]

EM: HMM example (more detail in coding example) —

[[[HMM forward model. applications.]]] [[[M step]]] [[[E step: (inefficient) formula]]] [[[E step: efficient way to compute via dynamic programming]]] [[[Example runthrough on toy data. applications.]]]

Learned E steps and Neural Networks — [[[why are E steps hard?]]] [[[how we might throw deep learning at the E step]]] [[[first taste of VAEs: architecture for image generation]]] [[[a remark on diffusion models; distribution-level losses and GANs]]] [[[a periodic table of ways to encode distributions as neural net-

works]]]

(Bonus) under the hood: ELBO bound, pingpong KL geometry — [[[useful math: interplay of logarithms and expectations]]] [[[ELBO bound and E,M steps]]] [[[KL divergence, over vs undershoot support, compression, and surprise]]] [[[exponential vs mixture families. M and E projections.]]] [[[ping-pong picture]]]

inference by sampling: MH

Challenge. MH Algorithm. Diffusion. — Okay, so we have a probabilistic model and some data. How do we do inference? We want an approximate posterior over part of the model conditioned on the data.

Visualizing MH. Proposals Matter —

MH: 3State Traffic example —

MH: HMM and GM example —

On Deep Learning and Noise —

deep generative models

Architecture —

VAEs and ELBO —

Interpreting Update Intuitively —

Output side Noise Model (e.g. square loss) —

Conditional VAEs —

Learning while Acting (and from Rewards)

rewards, actions, states. exploration

qualitative solutions to mdp, dynamic programming, bootstrap

reduce to supervision using q-learning

non-technical discussion of: curiosity, language, instruction, world-models

Prereq Helpers

high dimensions and linear algebra

linear algebra and approximation

Stand firm in your refusal to remain conscious during algebra. In real life, I assure you, there is no such thing as algebra.
— fran lebowitz

Linear algebra is the part of geometry that focuses on when a point is the origin, when a 'line' is a straight, and when two straight lines are parallel. Linear algebra thus helps us deal with the preceding pictures mathematically and automatically. The concept of 'straight lines' gives a simple, flexible model for extrapolation from known points to unknown points. That is intuitively why linear algebra will be crucial at every stage of 6.86x.

VISUALIZING HIGH DIMENSIONAL SPACES —

COLUMN VECTORS AND ROW VECTORS — The elements of linear algebra are **column vectors** and **row vectors**.° We have a set V of "column vectors". We're allowed to find V's zero vector and to add or scale vectors in V to get other vectors in V. V is the primary object we hold in our mind; perhaps, if we're doing image classification, then each column vector represents a photograph. We use the word "space" or "vector space" when talking about V to emphasize that we'd like to exploit visual intuitions when analyzing V. In short: we imagine each column vector as a point in space, or, if we like, as an arrow from the zero vector to that point.

← Though we represent the two similarly in a computer's memory, they have different geometric meanings. We save much anguish by remembering the difference.

Now, associated to V is the set of "row vectors". Under the hood, a row vector is a linear function from V to the real numbers $\mathbb{R}$. We imagine each row vector not as an arrow but as a "linear" heat map or altitude map that assigns to each point in V a numeric "intensity". We can visualize a row vector the same way makers of geographic maps do: using contours for where in V the row vector attains values $\cdots, -2, -1, 0, +1, +2, \cdots$. These will be a bunch of uniformly spaced parallel "planes". The spacing and orientation of the planes depends on and determines the row vector. In short, we imagine each row vector as a collection of parallel planes in space.

Informally: a column vector is a noun or thing whereas a row vector is a adjective or property. The degree to which a property holds on a thing (or a description is true of a thing) is gotten by evaluating the row vector on the column vector — remember, a row vector is a function, so we can do this evaluation. Geometrically, if a row vector is a bunch of planes and a column vector is an arrow, the two evaluate to a number: the number of planes that the arrow pierces. Intuitively, an example of a column vector might be "this particular photo of my pet cow"; an example of a row vector might be "redness of the left half (of the input photo)". If we evaluate this row vector on this column vector, then we get a number indicating how intensely true it is that the left half of that particular photo is red.

We can draw an analogy with syntax vs semantics. This pair of concepts pops up in linguistics, philosophy, circuit engineering, quantum physics, and more, but all we need to know is that: semantics is about things while syntax is about descriptions of things. The two concepts relate in that, given a set of things, we can ask for the set of all descriptions that hold true for all those things simultaneously. And if we have a set of descriptions, we can ask for the set of all things that satisfy all those descriptions simultaneously. These two concepts stand in formal opposition in the sense that: if we have a set of things and make it bigger, then the set of descriptions that apply becomes smaller. And vice versa. Then a column vector is an object of semantics. And a row vector is an object of

INNER PRODUCTS — Now, here is the key point: the engine behind generalization in machine learning (at least, the machine learning we'll do in Units 1 and 2; and less visibly but still truly in more than half of each of Units 3,4,5) is the ability to translate

between things and properties. If "my pet cow" is a thing, then "similar to my pet cow" is a property. The whole project of machine learning is to define and implement this word "similar to". When we define and implement well, our programs can generalize successfully from training examples to new, previously unseen situations, since they will be able to see which of the training examples the new situations are similar to. Since "similar to" transforms things to properties, the linear algebra math behind "similar to" is a function from column vectors to row vectors. This brings us to...

... inner products, aka kernels. An inner product is just a fancy word for a (linear) function from column vectors to row vectors. We actually demand that this linear function has two nice properties: FIRST, it should have an inverse. That is, it ought to be a two way bridge between column vectors and row vectors, allowing us to translate things to descriptions and vice versa. SECOND, it should be symmetric. This means that if we have two things, say "my pet cow" and "my pet raccoon", then the degree to which "my pet raccoon" has the property "is similar to my pet cow" ought to match the degree to which "my pet cow" has the property "is similar to my pet raccoon". Any invertible, symmetric, linear function from column vectors to row vectors is called an inner product. Kernel is a synonym here.

Beware: the same word, "kernel", has different meanings depending on context.

There are generally infinitely many inner products. Which one we choose changes the generalization properties of our machine learning program. Practically, if we are doing machine learning in a concrete situation, then we want to choose an inner product that reflects our human intuition and experience and domain knowledge about the right notion of "similarity" in that situation.

Any inner product P from column vectors to row vectors induces notions of length and angle. We just define a column vector v's length by $\sqrt{P(v)(v)}$. Call that quantity $\|v\|$. And we define the angle $\alpha(v, w)$ between two non-zero column vectors v,w by $P(v)(w) = \|v\| \cdot \|w\| \cdot \cos \alpha(v, w)$. We make these definitions so as to match the Pythagorean theorem from plane geometry. So once we choose which inner product we'll use (out of the infinitely many choices), we get the concepts of euclidean geometry for free. Immediately following from that definition of angle, we get that if two column vectors have vanishing inner product (i.e., if $P(v)(w) = 0$), then those vectors are at right angles (i.e. $\alpha(v)(w) = \pi/2$).

You can google up proofs of the Pythagorean theorem (many are quick and beautiful) if you wish to dig deeper.

Now, sometimes (but most of the time not), we are blessed in that we know more about our situation than just the space V of things. Specifically, V might come with a canonical basis. This just means that V comes marked with "the right" axes with respect to which we ought to analyze vectors in V. In this fortunate case, there is also a canonical inner product. It's called dot product. Again, I want to emphasize that a plain vector space doesn't come with a dot product. We need a basis for that.

The dot product is defined as follows. Say there are D axes and that the "unit" vectors along each axis (aka the basis vector) are named $(v_i : 0 \leq i < D)$. Then we define $P(v_j)(v_i) = (1 \text{ if } i = j \text{ else } 0)$ — the 1 expresses that each "unit" basis vector ought to have length 1. The 0 expresses that different basis vectors ought

to be at right angles to each other. This determines P on all inputs, by linearity: $P(\sum c'_j v_j)(\sum c_i v_i) = \sum_i c'_i c_i$ where c'_j, c_i are numbers. In short: given a basis, there is a unique inner product such that those basis elements all have length 1 and are at right angles to each other. We call that inner product the dot product.

FILL IN LINEAR DECISION BOUNDARY! (remark on featurization and argmax nonlinearities)

We may **evaluate** a row vector on a column vector. FILL IN A **dot product** is a way of translating between row and column vectors. FILL IN: DISCUSS GENERALIZATION; (DISCUSS ANGLE, TOO)

LINEAR MAPS —

SINGULAR VALUE DECOMPOSITION —

uncertainty and probability

probability and generalization

Can I just say Chris for one moment that I have a new theory about the brontosaurus. ... This theory goes as follows and begins now. All brontosauruses are thin at one end, much, much thicker in the middle and then thin again at the far end. That is my theory, it is mine, and belongs to me and I own it, and what it is too.
— john cleese

BELIEF AND BAYES —

THE KEY ABSTRACTION: AVERAGES —

THE KEY APPROXIMATION: INDEPENDENCE —

UNIFORM CONCENTRATION —

optimization and derivatives

calculus and optimization

The self is not something ready-made, but something in continuous formation through choice of action.
— john dewey

Throughout this section, X and Y will refer to two normed real vector spaces of finite dimension.

ASYMPTOTIC NOTATION — When analyzing algorithms or data, we often wish to consider extremes of the very small or the very large. Such thought experiments isolate how behaviors of interest depend on the variables we take to extreme values.

We say $f : X \to Y$ **is negligible compared to** $g : X \to Y$ **for sufficiently small inputs** when the ratio $\|f\|/\|g\|$ is tiny for small inputs — that is, when:

For any positive number ϵ
there exists a positive number δ so that,
whenever $0 < \|x\| < \delta$,
we also have $\|f(x)\| < \epsilon\|g(x)\|$.

The class of fs that are negligible compared to g we denote by

$$o(g)$$

or, when abusing notation, by $o(g(x))$ even though x isn't defined. This is **little-oh** notation.

For example, if p, q are positive real numbers then $|x|^p$ is negligible compared to $|x|^q$ if and only if $p < q$:

$$(x \mapsto |x|^3) \in o(x \mapsto |x|^2) \qquad (x \mapsto |x|^3) \notin o(x \mapsto |x|^4)$$

← The notion that "f *is negligible compared to* g *for sufficiently small inputs*" is the most important of a 2×2 grid of variants: we may change

sufficiently small $\rightsquigarrow$ sufficiently large

by replacing

"$0 < \|x\| < \delta$" $\rightsquigarrow$ "$\delta < \|x\|$"

and/or we may change

is negligible compared to
$\rightsquigarrow$ never overwhelms

by replacing

"For any positive number ϵ"
$\rightsquigarrow$ "There exists a positive number ϵ"

The class of fs that never ovewhelm g is called $O(g)$ — pronounced **big-Oh**. Clearly, $o(g) \subsetneq O(g)$. Confusingly, folks use the same notation $o(g), O(g)$ when considering small inputs and large inputs; which sense we mean should be clear from context.

Exercise: Is $\sin(x) \in o(1)$? How about $o(x)$?
Exercise: Is $\max(0, x) \in o(1)$? How about $o(x)$?
Exercise: Is $\log|x| \in o(1/x)$? (Ignore $x = 0$.)
Exercise: Is $\exp(-1/|x|) \in o(x)$? (Ignore $x = 0$.)

DERIVATIVES — If $f : X \to Y$ is a (potentially nonlinear) function between two finite-dimensional real vector spaces, we may wish to approximate f by a linear function (plus a constant). It is often unreasonable to ask that the approximation is good for all inputs; instead, we ask that the approximation is good near some specific input $x : X$:

$$f(x+h) \approx f(x) + (Df_x)(h)$$

Here, $(Df_x) : X \to Y$ is a linear map that translates changes h in f's input to changes $(Df_x)(h)$ in f's output. We want the approximation to be good for small h in the sense that the error vanishes faster than linearly as h shrinks:

$$\|f(x) + (Df_x)(h) - f(x+h)\| \in o(\|h\|)$$

Intuitively, Df_x exists when f varies smoothly.

INTEGRALS —

—

python, numpy, pytorch

G. (python) programming refresher

python setup

If I have not seen as far as others, it is because giants were standing on my shoulders.
— hal abelson

WHAT'S PYTHON? — Python is a programming language. Its heart is the **Python interpreter**, a computer program that takes a plain text file such as this two-liner° —

```
print('hello, world!')
print('cows and dogs are fluffy')
```

— and executes the instructions contained in that text file. We call these textual instructions are **source code**.

← The instruction `print` just displays some text onto our screen. For example, the first line of this program displays `hello, world!` onto our screen. This instruction doesn't rely on or activate any ink-on-paper printing machines.

The instructions have to be in a certain, extremely rigid format in order for the interpreter to understand and execute them. That's why they call Python a *language*: it has its own rigid grammar and its own limited vocabulary. If the interpreter encounters incorrectly formatted instructions — if even a single punctuation mark is missing or a — the interpreter will display a bit of text in addition to the word `Error` and immediately after abort its mission.°

← Adventure boldly when learning Python! It might feel catastrophic when you encounter an error and the interpreter 'dies'. But (unless you go out of your way to use special instructions we won't teach in class) these errors won't hurt your computer. There aren't any lasting effects. So **errors are hints, not penalties**. If you encounter an error, just modify your instructions to address that error, then run the interpreter again. Engage in a fast feedback cycle (*I'll try this... error... okay how about this... different error... hmm let's think...*) to learn to program well.

We'll use Python in 6.86x to instruct our computer to analyze data. The gigantic project of instructing a computer is a bit like teaching a person by mail. We never see them; we exchange only written words. They have never seen a horse, and yet we want to teach them to distinguish horses from zebras from donkeys from tapirs from rhinos. Though severely limited in their vocabulary and their ability to appreciate analogies and similarities, they are extraordinarily meticulous, patient, and efficient. That's what programming will be like.

At this point, you might have several questions:

Picking up the pen: How do I install and use the Python interpreter?

Writing sentences: What syntax rules must my instructions obey in order for the interpreter to understand that I want it do such-and-such task?

Composing essays: How do I organize large programs?

Teaching via mail: What instructions make the computer analyze data?

This and the next three sections address these four questions in turn.

WHICH THINGS WE'LL SET UP — Let's set up Python by installing the Python interpreter. Actually, I should say *a* Python interpreter: each of the many software tools we'll use in 6.86x has a zillion versions; it can get confusing tracking which versions coexist and which clash. We will use **Python version 3.8** throughout 6.86x.

Beyond the Python interpreter, there is an ecosystem of useful tools: machine learning modules that help us avoid reinventing the wheel; a rainbow of feature-rich text editors specialized for writing Python code; and a package manager called `conda` that eases the logistics nightmare of coordinating the versions we have of each tool.

SETUP ON WINDOWS — Mohamed, please fill this out (mention windows 10 and higher's 'linux subsystem?')

SETUP ON MACOS — Karene, please fill this out

SETUP ON LINUX — I'll assume we're working with Ubuntu 20.04. If you're on a different Linux, then similar steps should work — google search to figure out how.

CHECKING THE SETUP — Let's create a new plain text file containing this single line:°

```
print('hello, world!')
```

← This line contains **Python source code**. When we include Python source code in these notes, we will format it for readability, e.g. by making key parts of it bold. However, this depiction of source code is supposed to represent *plain text*.

We can name the file whatever we want — say, `greetings.py`. Then in your terminal (navigate to the same directory as the file and) enter this command:

```
python3 greetings.py
```

A new line should appear in your terminal:

```
hello, world!
```

After that line should be another shell prompt.

Now append three lines to the file `greetings.py` so that its contents look like:

```
print('hello, world!')
fahr = int(input('please enter a number... '))
celc = int((fahr - 32.0) * 5.0/9.0)
print('{} fahrenheit is roughly {} celcius'.format(fahr, celc))
```

Again, enter the following command in your terminal:

```
python3 greetings.py
```

Two new lines should appear in your terminal:

```
hello, world!
please enter a number...
```

Enter some number — say, 72; a new line should appear in your terminal so that the overall session looks something like:

```
hello, world!
please enter a number... 72
72 fahrenheit is roughly 22 celcius
```

After that line should be another shell prompt.

If you did the above but something different from the predicted lines appeared, that means something is amiss with Python setup. Please let us know in this case! We can try to help fix.

elements of programming

Displace one note and there would be diminishment, displace one phrase and the structure would fall.
— antonio salieri, on wolfgang mozart's music, as untruthfully portrayed in *Amadeus*

STATE AND CONTROL FLOW —

ROUTINES — Say there's an operation we do a lot. Maybe it's fahrenheit-to-celcius conversion. Instead of typing out the formula each time, we can *name* that formula and then invoke it by typing its name. Like so:

```
celc_from_fahr = lambda f: (f-32.)*(5./9.)
print(celc_from_fahr( 32.)) # prints   0.
print(celc_from_fahr(212.)) # prints 100.
```

Introducing such abstraction aligns the code's structure with our mental conceptual structure, yielding two related advantages: (a) it makes our source code easier to reason about (easier to read, easier to check correctness of, easier to change); (b) hierarchies of such formulae allows us to express concisely what otherwise would be an intractably long program.

To illustrate (a)

```
celc_from_fahr = lambda f: (f-32.)*(5./9.)
kelv_from_celc = lambda c: c - 273.15
kelv_from_fahr = lambda f: kelv_from_celc(celc_from_fahr(f))
kelv_from_kelv = lambda k: k
kelv_from_str  = lambda s: {'K':kelv_from_kelv,
                            'C':kelv_from_celc,
                            'F':kelv_from_fahr}[s[-1]](float(s[:-1]))
kelvs_from_csv = lambda ss: max(kelv_from_str(s.strip())
                                for s in ss.split(','))
#
avg        = lambda ts: sum(ts)/len(ts)
avg_square = lambda ts: average([t**2 for t in ts])
max_min_avg= lambda ts: (max(ts), min(ts), average(ts))
fancy_stats= lambda ts: (lambda mx,mn,av: {'max'     :mx,
                                           'min'     :mn,
                                           'avg':    :av,
                                           'variance':avg_square(ts)-av**2,
                                           'skew'    :av-(mn+mx)/2})(max_min_avg(ts))
#
csv = '50␣C,␣40␣F,␣200␣K,␣90␣F,␣80␣F,␣70␣F'
print(max_temp(kelvs_from_csv(csv)))
```

For more complicated routines, we can use this notation:

```
def celc_from_fahr(f):
  return (f-32.)*(5./9.)
```

We can use multiple lines and include assignments and control flow:

```
def celc_from_fahr(f):
  shifted = f-32.
  if shifted < 0:
    print('brrr!')
  return shifted*(5./9.)
```

Sometimes we want to

DATA IN AGGREGATE — lists and numpy arrays dictionaries comprehensions functional idioms

INPUT/OUTPUT — print formatting and flushing basic string manipulations (strip, join, etc) file io command line arguments example: read csv random generation and seeds??

how not to invent the wheel

OS, NUMPY, MATPLOTLIB, ETC — matplotlib.plot numpy os package managers etc

ARRAYS AKA TENSORS —

PYTORCH IDIOMS —

GIT FOR VERSION CONTROL —

```
def get_random_number():
    return 4 # chosen by a fair dice roll
             # guaranteed to be random
```

— randall munroe, translated by sam to Python

how to increase confidence in correctness

So little of what could happen does happen.
— salvador dalí

DEDUCTION: —

INDUCTION: ISOLATION AND BISECTION —

PERIODIC TABLE OF COMMON ERRORS —