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 h_t misclassifies a particular $(x_i, y_i) \in \mathcal{S}$, then we'd like h_{t+1} to be like h_t but nudged toward accurately classifying (x_i, y_i).°

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

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

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

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

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

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

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

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

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

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

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

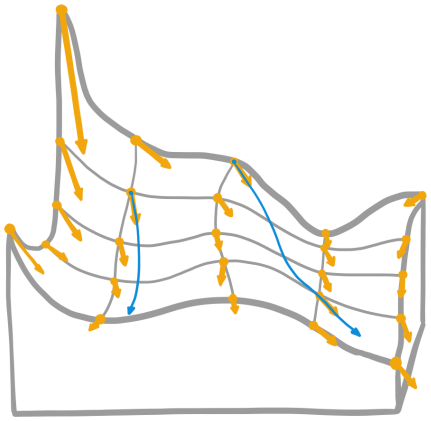

Figure 17: An intuitive picture of gradient descent. Vertical axis is training loss and the other two axes are weight-space. Warning: though logistic loss (and the L2 regularizer) is smooth, perceptron and hinge loss (and the L1 regularizer) have jagged angles or 'joints'. Also, all five of those functions are convex. The smooth, nonconvex picture here is most apt for deep learning (Unit 3).

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 $\sigma(-\vec{w} \cdot \vec{x})$:

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

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

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

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

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

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

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

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

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

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

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

Then we can finally write gradient descent:

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

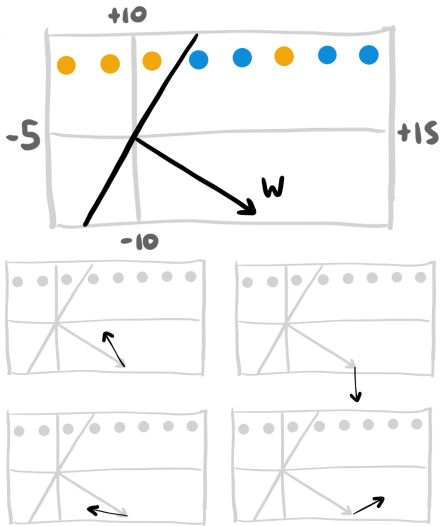

Figure 18: Food For Thought: Shown is a small training set (originally 1-D but featurized using the bias trick) and our current weight vector. Here blue is the positive label. *Which direction will the weight vector change if we do one step of (full-batch) gradient descent?* Four options are shown; one is correct.

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°

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

instead of $\lambda w \cdot w/2 + \sum_k \log(1/\sigma(y_k w \cdot \varphi(x_k)))$. We can do this by gradient descent.

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

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

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

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

or

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

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

← Food For Thought: 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?

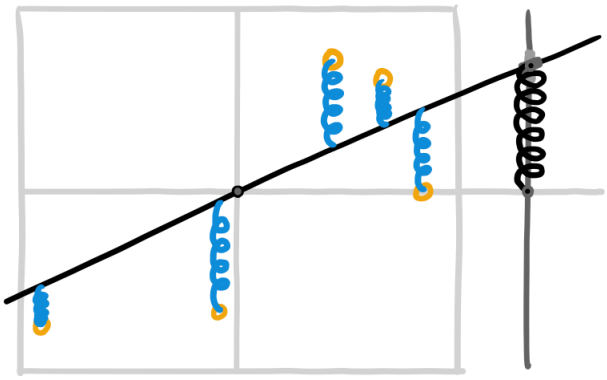

Figure 19: Here's an illuminating physical analogy. The least-squares loss says we want decision function values d to be close to the true ys. So we can imagine hooking up a stretchy spring from each training point (x_i, y_i) to our **predictor line (or hyperplane)**. Bolt that line to the origin, but 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 an 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.

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