smarter optimization

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

←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 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?

```
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):°

←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.

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

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):

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

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 + ....`