CIS 421/521:
ARTIFICIAL INTELLIGENCE

Stochastic Gradient Descent

Jurafsky and Martin Chapter 5

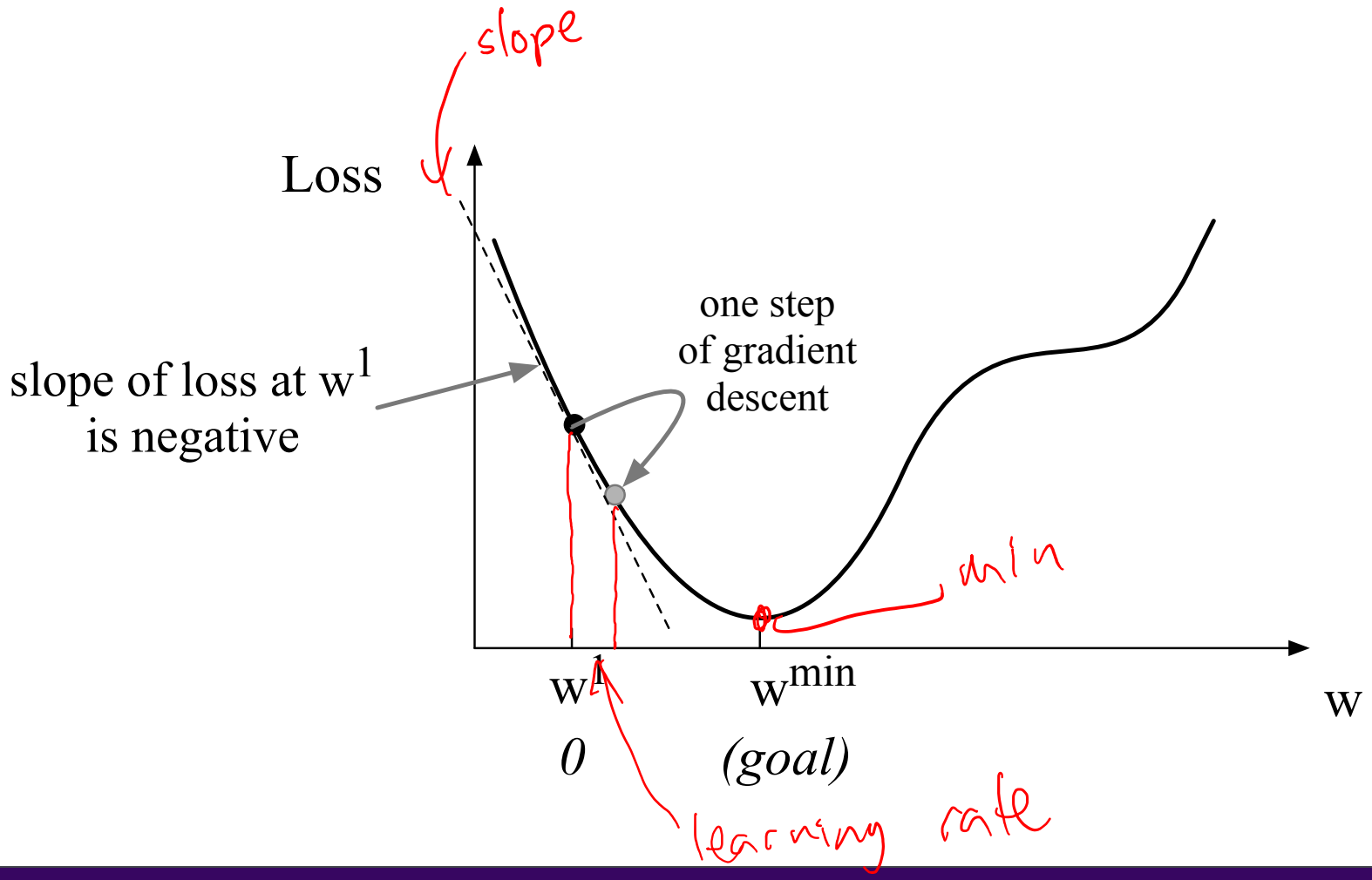
Stochastic gradient descent algorithm

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

where: L is the loss function
f is a function parameterized by θ
x is the set of training inputs $x^{(1)}$, $x^{(2)}$,..., $x^{(n)}$
y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$,..., $y^{(n)}$

$\theta \leftarrow 0$
repeat T times
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
  Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
  Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$) from the true output $y^{(i)}$?
  $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss ?
  $\theta \leftarrow \theta - \eta\, g$ # go the other way instead
return θ

Iteratively find minimum



How much should we update the parameter by?

The magnitude of the amount to move in gradient descent is the value of the slope weighted by a learning rate η.

A higher/faster learning rate means that we should move *w* more on each step.

$$w^{t+1} = w^t - \eta \frac{d}{dw} f(x; w)$$

time t+1

"eta"

slope

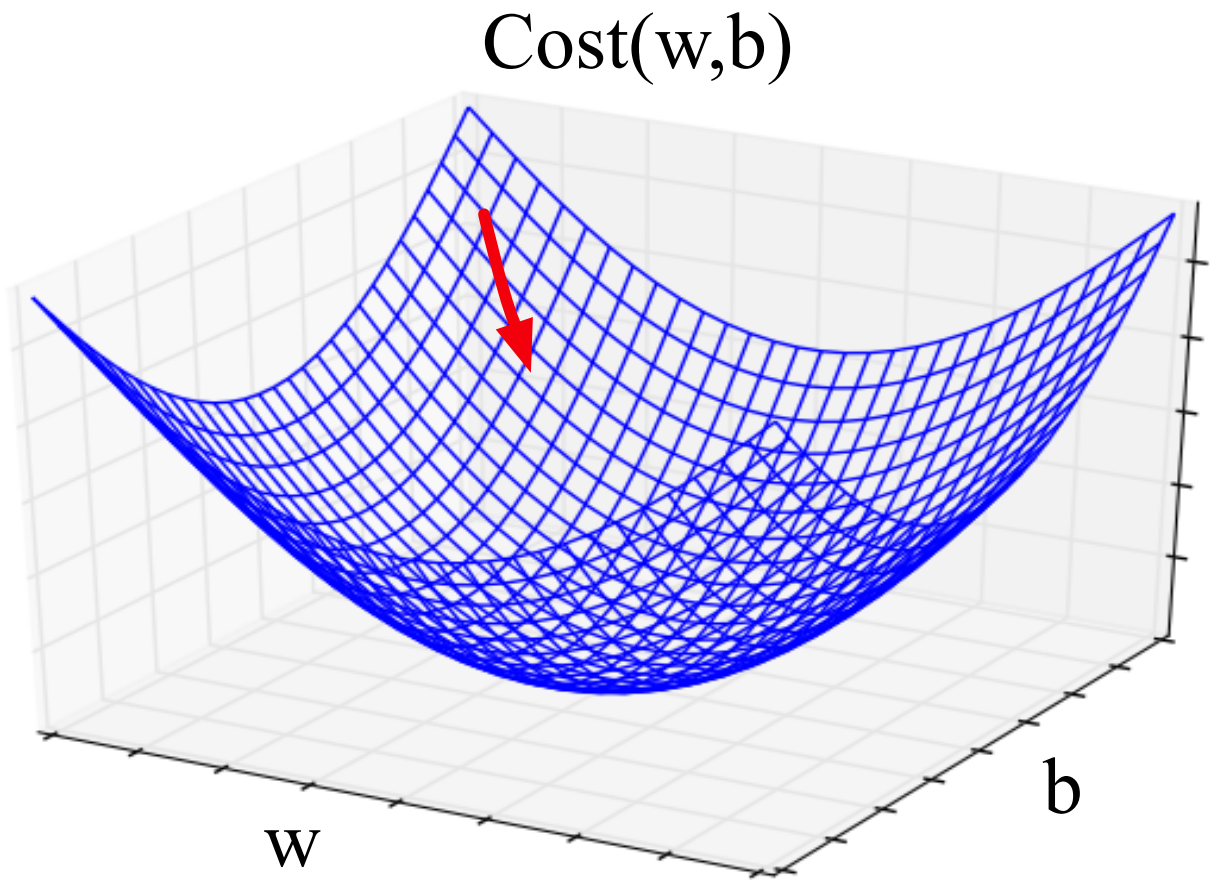
new weight

old weight

minus

learning rate * derivative of fn.

Many dimensions



Updating each dimension w_i

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \end{bmatrix}$$

y

our model's prediction for input x given parameters θ

The final equation for updating θ based on the gradient is

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y)$$

Learning rate

The Gradient

To update θ, we need a definition for the gradient $\nabla L(f(x;\theta), y)$.

For logistic regression, the cross-entropy loss function is:

$$L_{CE}(w,b) = -[y \log \sigma(w \cdot x + b) + (1-y) \log (1 - \sigma(w \cdot x + b))]$$

The derivative of this function for one observation vector x for a single weight w_j is

$$\frac{\partial L_{CE}(w,b)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j$$

truth

our model's prediction

* value of feature j

The gradient is a very intuitive value: the difference between the true y and our estimate for x, multiplied by the corresponding input value x_j .

Average Loss

$$Cost(w,b) = \frac{1}{m}\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)})$$

$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma\left(w \cdot x^{(i)} + b\right) + \left(1 - y^{(i)}\right)\log\left(1 - \sigma\left(w \cdot x^{(i)} + b\right)\right)$$

This is what we want to minimize!!

The Gradient

The loss for a batch of data or an entire dataset is just the average loss over the *m* examples

$$Cost(w,b) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma\left(w \cdot x^{(i)} + b\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(w \cdot x^{(i)} + b\right)\right)$$

The gradient for multiple data points is the sum of the individual gradients:

$$\frac{\partial Cost(w,b)}{\partial w_j} = \sum_{i=1}^{m} [\sigma\left(w \cdot x^{(i)} + b\right) - y^{(i)}] x_j^{(i)}$$

Worked example

Let's walk though a single step of the gradient descent algorithm. We'll use a simple sentiment classifier with just 2 features, and 1 training instance where the correct value is $y = 1$ (this is a positive review).

$$x_1 = 3 \quad \text{(count of positive lexicon words)}$$

$$x_2 = 2 \quad \text{(count of positive negative words)}$$

The initial weights and bias in θ^0 are all set to 0, and the initial learning rate η is 0.1:

$$w_1 = w_2 = b = 0$$

$$\eta = 0.1$$

The single update step requires that we compute the gradient, multiplied by the learning rate:

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

Worked example

The derivative of this function for **a single training example** *x* for a single weight w_j is

$$\frac{\partial L_{CE}(w,b)}{\partial w_j} = [s(w \cdot x + b) - y]x_j$$

The gradient vector has 3 dimensions, for w_1, w_2, and b.
For our input, $x_1 = 3$ and $x_2 = 2$

$x_2 = 2$

$$\nabla_{w,b} = \begin{bmatrix} \frac{\partial L_{CE}(w,b)}{\partial w_1} \\ \frac{\partial L_{CE}(w,b)}{\partial w_2} \\ \frac{\partial L_{CE}(w,b)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(w \cdot x + b) - y)x_1 \\ (\sigma(w \cdot x + b) - y)x_2 \\ \sigma(w \cdot x + b) - y \end{bmatrix} = \begin{bmatrix} (\sigma(0) - 1)x_1 \\ (\sigma(0) - 1)x_2 \\ \sigma(0) - 1 \end{bmatrix} = \begin{bmatrix} -0.5x_1 \\ -0.5x_2 \\ -0.5 \end{bmatrix} = \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix}$$

Worked example

Now that we have a gradient $\nabla_{w,b}$, we compute the new parameter vector θ^1 by moving θ^0 in the opposite direction from the gradient:

$$\theta^1 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

So after one step of gradient descent, the weights have shifted to be:

w_1 = 0.15, w_2 = 0.1, and b= .05

Mini-batch training

Stochastic gradient descent chooses a **single random example** at a time and updates its weights on that example. As a result the updates can fluctuate.

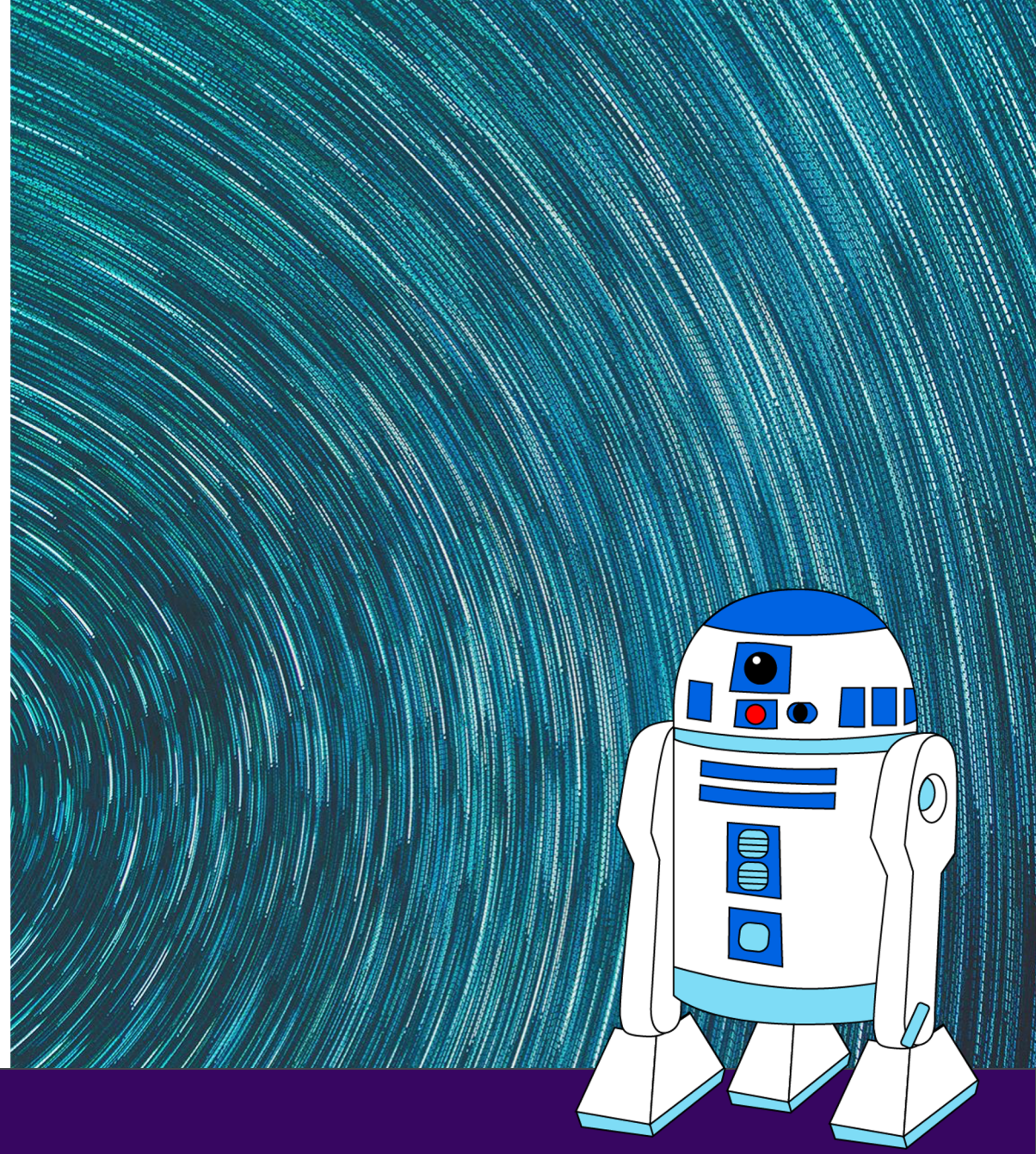
An alternate is **batch training,** which computes the gradient over the **entire dataset**. This gives a much better estimate of which direction to move the weights but takes a long time to compute.

A commonly used compromise is **mini-batch training**, where we train on a small batch. The batch size can be 512 or 1024, often selected based on computational resources, so that all examples in the mini-batch can be processed in parallel. The loss is then accumulated.

CIS 421/521:
ARTIFICIAL INTELLIGENCE

Logistic Regression – Wrap Up

Jurafsky and Martin Chapter 5



Penn Engineering
University of Pennsylvania

Regularization

Overfitting is a problem with many machine learning models. Overfitting results in poor generalization and poor performance on unseen test set.

In logistic regression, if a feature only occurs in one class then it will get a **high weight**. Sometimes we are just modelling noisy factors that just accidentally correlate with the class.

Regularization is a way to penalize large weights. A regularization term is added to the loss function.

Lasso regression uses L1 regularization
Ridge regression uses L2 regularization

Multinomial logistic regression

Instead of binary classification, we often want more than two classes. For sentiment classification we might extend the class labels to be **positive, negative**, and **neutral**.

We want to know the probability of y for each class $c \in C$, $p(y = c|x)$.

To get a proper probability, we will use a **generalization of the sigmoid function** called the **softmax function.**

$$\text{softmax}(z_i) = \frac{e^{z_j}}{\sum_{j=1}^{k} e^{z_j}} \; 1 \leq i \leq k$$

Softmax

The softmax function takes in an input vector $z = [z_1, z_2, \ldots, z_k]$ and outputs a vector of values normalized into probabilities.

$$\text{softmax}(z) = \left[\frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}, \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}}, \cdots, \frac{e^{z_k}}{\sum_{i=1}^{k} e^{z_i}}\right]$$

For example, for this input:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

Softmax will output:

$$[0.056, 0.090, 0.007, 0.099, 0.74, 0.010]$$