CS447: Natural Language Processing

http://courses.engr.illinois.edu/cs447

Lecture 5: Logistic Regression

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Part 1: Review and Overview

Probabilistic classifiers

We want to find the *most likely* class y for the input $\mathbf{x}$:

$$y^* = \text{argmax}_y \; P(Y = y \,|\, \mathbf{X} = \mathbf{x})$$

$P(Y = y \,|\, \mathbf{X} = \mathbf{x})$:

The probability that the class label is y when the input feature vector is $\mathbf{x}$

$y^* = \text{argmax}_y \; f(y)$

Let y^* be the y that maximizes $f(y)$

Modeling $P(Y|X)$ with Bayes Rule

Bayes Rule relates $P(Y|X)$ to $P(X|Y)$ and $P(Y)$:

$$\underbrace{P(Y|X)}_{\text{Posterior}} = \frac{P(Y,X)}{P(X)}$$

$$= \frac{P(X|Y)P(Y)}{P(X)}$$

$$\propto \overbrace{P(X|Y)}^{\text{Likelihood}} \overbrace{P(Y)}^{\text{Prior}}$$

Bayes rule: The **posterior** $P(Y \mid X)$ is proportional to the **prior** $P(Y)$ times the **likelihood** $P(X|Y)$

$P(Y|X)$ with Bayes Rule

Posterior $P(Y \mid X)$
Probability of the label Y ***after*** having seen the data X

Bayes Rule relates $P(Y|X)$ to $P(X|Y)$ and $P(Y)$:

$$P(Y|X) = \frac{P(Y,X)}{P(X)}$$

Posterior

$$= \frac{P(X|Y)P(Y)}{P(X)}$$

$$\propto P(X|Y)P(Y)$$

Likelihood

Prior

Likelihood $P(X \mid Y)$
Probability of the data X according to class Y

Prior $P(Y)$
Probability of the label Y ***independent of*** the data X

Bayes rule: The **posterior** $P(Y \mid X)$ is proportional to the **prior** $P(Y)$ times the **likelihood** $P(X|Y)$

Using Bayes Rule for our classifier

$$y^* = \text{argmax}_y P(Y \mid \mathbf{X})$$

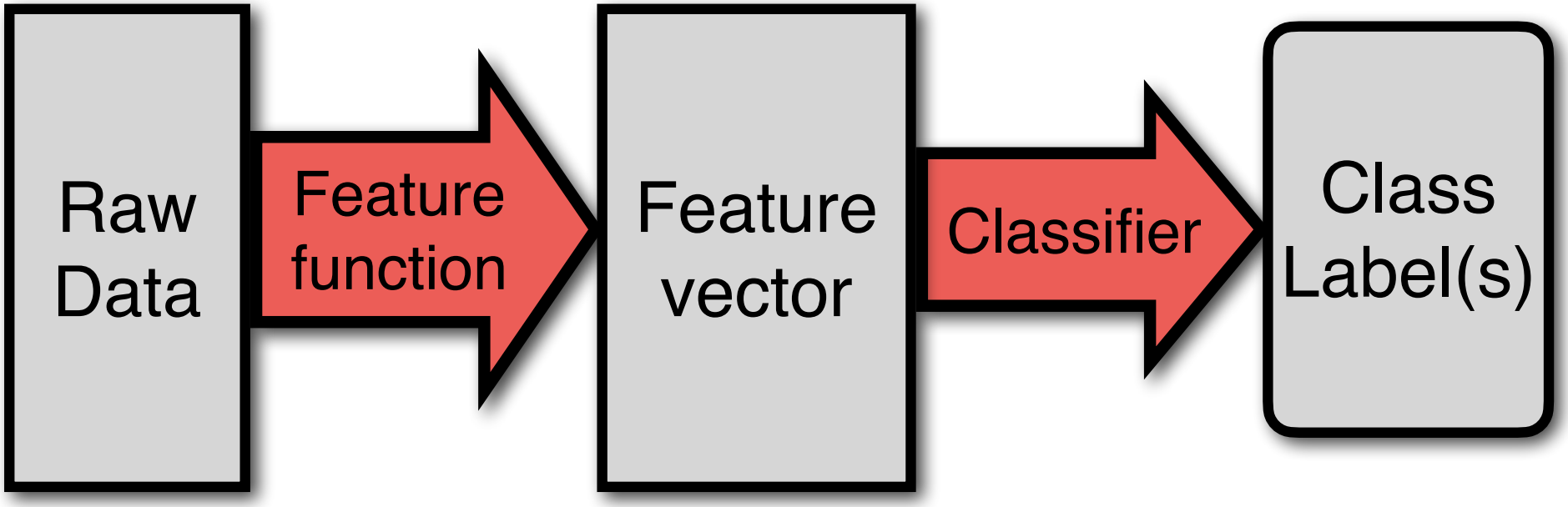
$$= \text{argmax}_y \frac{P(\mathbf{X} \mid Y)P(Y)}{P(\mathbf{X})}$$

$$= \text{argmax}_y P(\mathbf{X} \mid Y)P(Y)$$

[Bayes Rule]

[$P(X)$ doesn't change argmax_y]

Classification more generally



Before we can use a classifier on our data, we have to map the data to **"feature" vectors**

Feature engineering as a prerequisite for classification

To talk about classification mathematically, we assume each input item is represented as a **'feature' vector** $\mathbf{x} = (x_1 \ldots x_N)$

- Each element in **x** is one feature.
- The number of elements/features N is fixed, and may be very large.
- **x** has to capture *all* the information about the item that the classifier needs.

But the raw data points (e.g. documents to classify) are typically not in vector form.

Before we can train a classifier, we therefore have to first define a suitable **feature function** that maps raw data points to vectors.

In practice, **feature engineering** (designing suitable feature functions) is very important for accurate classification.

Probabilistic classifiers

A **probabilistic classifier** returns the *most likely* class y^* for input $\mathbf{x}$:

$$y^* = \text{argmax}_y P(Y = y \mid \mathbf{X} = \mathbf{x})$$

[Last class:] **Naive Bayes** uses Bayes Rule:

$$y^* = \text{argmax}_y P(y \mid \mathbf{x}) = \text{argmax}_y P(\mathbf{x} \mid y) P(y)$$

Naive Bayes models the joint distribution of the class and the data:

$$P(\mathbf{x} \mid y) P(y) = P(\mathbf{x}, y)$$

Joint models are also called **generative** models because we can view them as stochastic processes that *generate* (labeled) items:

Sample/pick a label y with $P(y)$, and then an item $\mathbf{x}$ with $P(\mathbf{x} \mid y)$

[Today:] **Logistic Regression** models $P(y \mid \mathbf{x})$ directly

This is also called a **discriminative** or **conditional** model, because it only models the probability of the class given the input, and not of the raw data itself.

Key questions for today's class

What do we mean by **generative vs. discriminative** models/classifiers?

Why is it difficult to **incorporate complex features** into a generative model like Naive Bayes?

How can we use (standard or multinomial) **logistic regression** for (binary or multiclass) classification?

How can we train logistic regression models with **(stochastic) gradient descent**?

Today's class

Part 1: Review and Overview

Part 2: From generative to discriminative classifiers
(Logistic Regression
and Multinomial Regression)

Part 3: Learning Logistic Regression Models
with (Stochastic) Gradient Descent

Reading: Chapter 5 (Jurafsky & Martin, 3rd Edition)

Part 2: From Generative to Discriminative Probability Models

Probabilistic classifiers

A **probabilistic classifier** returns the *most likely* class y^* for input $\mathbf{x}$:

$$y^* = \text{argmax}_y P(Y = y \mid \mathbf{X} = \mathbf{x})$$

[Last class:] **Naive Bayes** uses Bayes Rule:

$$y^* = \text{argmax}_y P(y \mid \mathbf{x}) = \text{argmax}_y P(\mathbf{x} \mid y) P(y)$$

Naive Bayes models the joint distribution of the class and the data:

$$P(\mathbf{x} \mid y) P(y) = P(\mathbf{x}, y)$$

Joint models are also called **generative** models because we can view them as stochastic processes that *generate* (labeled) items:

Sample/pick a label y with $P(y)$, and then an item $\mathbf{x}$ with $P(\mathbf{x} \mid y)$

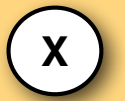
[Today:] **Logistic Regression** models $P(y \mid \mathbf{x})$ directly

This is also called a **discriminative** or **conditional** model, because it only models the probability of the class given the input, and not of the raw data itself.

(Directed) Graphical Models

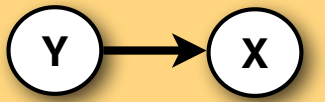
Graphical models are a visual notation
for probability models.

Each **node** represents a **distribution**
over one random variable:

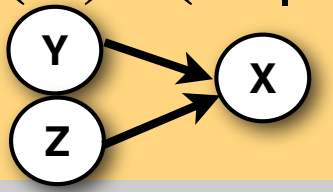
$P(X)$: 

Arrows represent **dependencies** (i.e. what other random variables the current node is conditioned on)

$P(Y)P(X \mid Y)$



$P(Y)P(Z)P(X \mid Y,Z)$

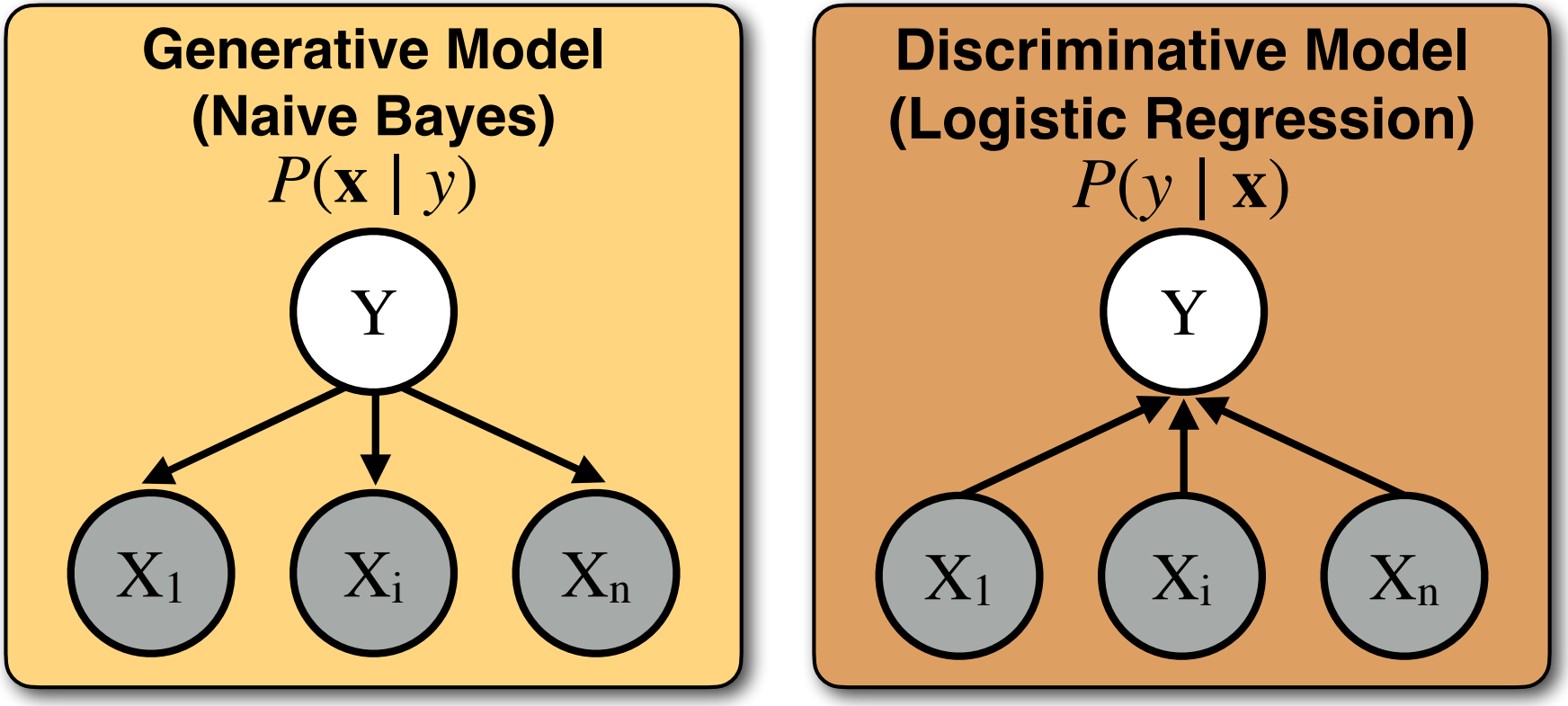


Generative vs Discriminative Models

In classification:

— The data $\mathbf{x} = (x_1, \ldots, x_n)$ is observed (shaded nodes).

— The label y is hidden (and needs to be inferred)



How do we model $P(Y = y \mid \mathbf{X} = \mathbf{x})$ such that we can compute it for *any* $\mathbf{x}$?

We've probably never seen any *particular* $\mathbf{x}$ that we want to classify at test time.

Even if we could define and compute **probability distributions**

$P(Y = y \mid X_i = x_i)$

with $\Sigma_{y_j \in Y} P(Y = y_j \mid X_i = x_i) = 1$ Good! $P(Y)$ sums to 1

for **any *single* feature** $x_i \in \mathbf{x} = (x_1, \ldots, x_i, \ldots, x_n)$…

….we can't just multiply these probabilities together to get ***one* distribution** over all $y_j \in Y$ for a given $\mathbf{x}$

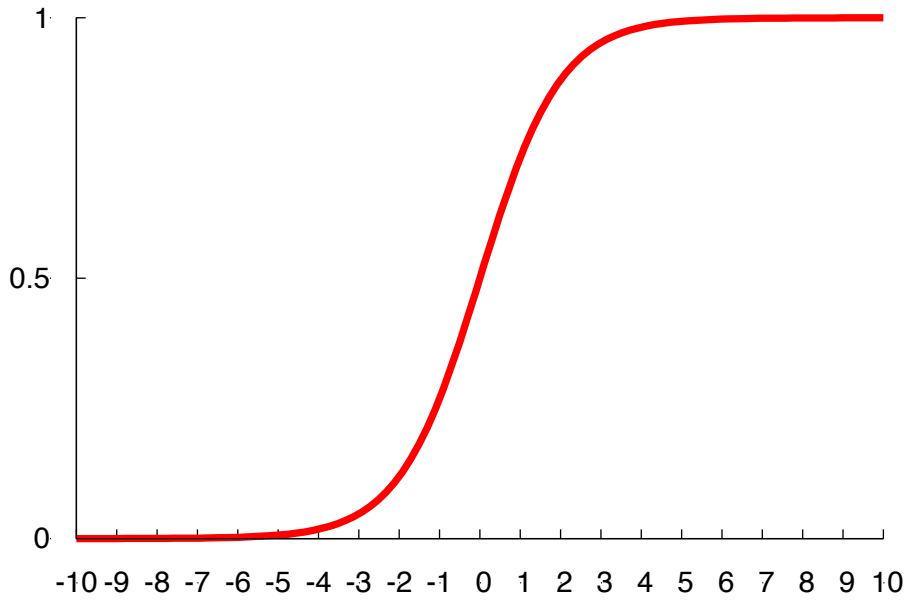
$$P(Y = y \mid \mathbf{X} = \mathbf{x}) := \sum_{y_j \in Y} \left[\prod_{i=1 \ldots n} P(Y = y_j \mid X_i = x_i) \right] < 1$$

Bad! $P(Y)$ does not sum to 1

The sigmoid function $\sigma(x)$

The **sigmoid function** $\sigma(x)$ maps any real number x to the range (0,1):

$$\sigma(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}$$



Using $\sigma()$ with feature vectors $\mathbf{x}$

We can use the sigmoid $\sigma()$ to express a Bernoulli distribution

Coin flips: $P(\text{Heads}) = \sigma(x)$ and $P(\text{Tails}) = 1 - P(\text{Heads}) = 1 - \sigma(x)$

But to use the sigmoid $\sigma()$ for binary classification,
we need to model **the *conditional* probability** $P(Y \in \{0,1\} \mid \mathbf{x} = \mathbf{X})$
such that it depends on the particular feature vector $\mathbf{x} \in \mathbf{X}$

Also: We don't know how **important each feature** (element) x_i
of $\mathbf{x} = (x_1, \ldots, x_n)$ for our particular classification task is…
… and we need to feed ***a single real number*** **into** $\sigma()$!

Solution: Assign (learn) a vector of **feature weights** $\mathbf{f} = (f_1, \ldots, f_n)$
and compute $\mathbf{fx} = \sum_{i=1}^{n} f_i x_i$ to obtain a single real, and then $\sigma(\mathbf{fx})$

P(Y | **X**) with Logistic Regression: Binary Classification

Task: Model $P(y \in \{0,1\} \mid \mathbf{x})$
for any input (feature) vector $\mathbf{x} = (x_1, \ldots, x_n)$

Idea: Learn feature weights $\mathbf{w} = (w_1, \ldots, w_n)$ (and a bias term b)
to capture how important each feature x_i is for predicting $y = 1$

For **binary** classification ($y \in \{0,1\}$),
(standard) logistic regression uses the sigmoid function:

$$P(Y=1 \mid \mathbf{x}) = \sigma(\mathbf{wx} + b) = \frac{1}{1 + \exp(-(\mathbf{wx} + b))}$$

Parameters to learn: one feature weight vector $\mathbf{w}$ and one bias term b

What about **multi-class classification**?

Now we need to model $P(Y \mid \mathbf{X})$ such that…

… The probability of any class y_i depends on j and $\mathbf{x}$:

-> Define **class-specific feature weights** $\mathbf{f}_j$: $\mathbf{f}_j\mathbf{x}$

… The probability of any one class y_j (for any input $\mathbf{x}$) is positive: $\forall_{\mathbf{x}\in\mathbf{X}} \forall_{j\in\{1...K\}} : P(Y = y_j \mid \mathbf{X} = \mathbf{x}) > 0$

-> **Exponentiate** $\mathbf{f}_j\mathbf{x}$: $\exp(\mathbf{f}_j\mathbf{x})$

… And the probabilities of all classes y_j (for each input $\mathbf{x}$) sum to one: $\forall_{\mathbf{x}\in\mathbf{X}} : \Sigma_{j=1..K} P(Y = y_j \mid \mathbf{X} = \mathbf{x}) = 1$

-> **Renormalize** $\exp(\mathbf{f}_j\mathbf{x})$: $P(Y = y_i \mid \mathbf{X} = \mathbf{x}) = \frac{\exp(\mathbf{f}_j\mathbf{x})}{\sum_k \exp(\mathbf{f}_k\mathbf{x})}$

$P(Y \mid \mathbf{X})$ with Logistic Regression: Multiclass Classification

Task: Model $P(\ y \in \{y_1, \ldots, y_K\} \mid \mathbf{x})$
for any input (feature) vector $\mathbf{x} = (x_1, \ldots, x_n)$

Idea: Learn feature weights $\mathbf{w}_j = (w_{1j}, \ldots, w_{nj})$ (and a bias term b_j) to capture how important each feature x_i is for predicting class y_j

For **multiclass** classification ($y \in \{0,1,...,K\}$), multinomial logistic regression uses the **softmax** function:

$$P(\ Y{=}y_j \mid \mathbf{x}\ ) = \text{softmax}(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)} = \frac{\exp(\ -(\mathbf{w}_j\mathbf{x} + b_j))}{\sum_{k=1}^{K} \exp(\ -(\mathbf{w}_k\mathbf{x} + b_k))}$$

Parameters to learn: one feature weight vector $\mathbf{w}_j$ and one bias term b_j *per class*

The **softmax** function

The **softmax** function turns *any* vector of reals $\mathbf{z} = (z_1, \ldots, z_n)$ into a discrete probability distribution $\mathbf{p} = (p_1, \ldots, p_n)$ where $\forall_{j \in \{1,\ldots,n\}}: 0 < p_j < 1$ and $\Sigma_{j=1}^{n} p_j = 1$

$$p_j = \text{softmax}(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}$$

Logistic regression applies the softmax to a linear combination of the input features $\mathbf{x}$: $\mathbf{z} = \mathbf{fx}$

Models based on logistic regression are also known as Maximum Entropy (**MaxEnt**) models

We will see the softmax again when we talk about **neural nets,** but there the input is typically a much more complex, nonlinear function of the input features.

NB: Binary logistic regression is just a special case of multinomial logistic regression

Binary logistic regression needs a distribution over $y \in \{0,1\}$:

$$P(Y=1 \mid \mathbf{x}) = \frac{1}{1+\exp(-(\mathbf{w}\mathbf{x}+b))}$$

$$P(Y=0 \mid \mathbf{x}) = \frac{\exp(-(\mathbf{w}\mathbf{x}+b))}{1+\exp(-(\mathbf{w}\mathbf{x}+b))} = 1 - P(Y=1 \mid \mathbf{x})$$

Compare with **Multinomial logistic regression** over $y \in \{0,1\}$:

$$P(Y=1 \mid \mathbf{x}) = \frac{\exp(-(\mathbf{w}_1\mathbf{x}+b_1))}{\exp(-(\mathbf{w}_1\mathbf{x}+b_1)) + \exp(-(\mathbf{w}_0\mathbf{x}+b_0))}$$

$$P(Y=0 \mid \mathbf{x}) = \frac{\exp(-(\mathbf{w}_0\mathbf{x}+b_0))}{\exp(-(\mathbf{w}_1\mathbf{x}+b_1)) + \exp(-(\mathbf{w}_0\mathbf{x}+b_0))}$$

➜ Binary logistic regression is a special case of multinomial logistic regression over two classes with $\exp(-(\mathbf{w}_1\mathbf{x}+b_1)) = 1$
(i.e. where $\mathbf{w}_1$ is set to the null vector and $b_1 := 0$)

Using Logistic Regression

How do we create a (binary) logistic regression classifier?

1) **Feature design**:
Decide how to map raw inputs to feature vectors $\mathbf{x}$

2) **Training**:
Learn parameters $\mathbf{w}$ and b on training data

Feature Design:
From raw inputs to feature vectors $\mathbf{x}$

Feature design for generative models (Naive Bayes):

— In a generative model, we have to learn a model for $P(\mathbf{x} \mid y)$.

— Getting a proper distribution ($\sum_{\mathbf{x}} P(\mathbf{x} \mid y) = 1$) is difficult

— NB assumes that the features (elements of $\mathbf{x}$) are independent* and defines $P(\mathbf{x} \mid y) = \prod_i P(x_i \mid y)$ via a multinomial or Bernoulli

(*more precisely, conditionally independent given y)

— Different kinds of feature values (boolean, integer, real) require different kinds of distributions $P(x_i \mid y)$ (Bernoulli, multinomial, etc.)

Feature Design:
From raw inputs to feature vectors $\mathbf{x}$

Feature design for conditional models (Logistic Regression):

- In a conditional model, we only have to learn $P(y \mid \mathbf{x})$
- It is much easier to get a proper distribution ($\sum_{j=1..K} P(y_j \mid \mathbf{x}) = 1$)
- We don't need to assume that our features are independent
- Any numerical feature x_i can be used *directly* to compute $\exp(w_{ij}x_i)$

Useful features that are *not* independent

Different features can *overlap* in the input

(e.g. we can model both unigrams and bigrams, or overlapping bigrams)

Features can capture *properties* of the input

(e.g. whether words are capitalized, in all-caps, contain particular [classes of] letters or characters, etc.)

This also makes it easy to use predefined dictionaries of words (e.g. for sentiment analysis, or gazetteers for names):
Is this word "positive" ('*happy*') or "negative" ('*awful*')?

Is this the name of a person ('Smith') or city ('Boston') [it may be both ('Paris')]

Features can capture *combinations* of properties

(e.g. whether a word is capitalized *and* ends in a full stop)

We can use the outputs of other classifiers as features

(e.g. to combine weak [less accurate] classifiers for the same task,
or to get at complex properties of the input that require a learned classifier)

Feature Design and Selection

How do you specify features?

We can't manually enumerate 10,000s of features
(e.g. for every possible bigram: "*an apple*", …, "*zillion zebras*")

Instead we use **feature templates** that define what type of feature we want to use
(e.g. "*any pair of adjacent words that appears >2 times in the training data*")

How do you know which features to use?

Identifying useful sets of feature templates requires **expertise** and a lot of **experimentation** (e.g. ablation studies)
Which specific set of feature (templates) works well depends very much on the particular classification task and dataset.

Feature selection methods prune useless features automatically. This reduces the number of weights to learn.
(e.g. '*of the*' may not be useful for sentiment analysis, but '*very cool*' is)

Part 3: Training Logistic Regression Models with (Stochastic) Gradient Descent

Learning parameters **w** and *b*

Training objective: Find parameters **w** and *b* that "capture the training data D_{train} as well as possible"

More formally (and since we're being probabilistic):

Find **w** and *b* that assign the largest possible conditional probability to the labels of the items in D_{train}

$$(\mathbf{w}^*, b^*) = \text{argmax}_{(\mathbf{w},b)} \prod_{(\mathbf{x}_i,y_i)\in D_{train}} P(y_i \mid \mathbf{x}_i)$$

⇒ Maximize $P(1 \mid \mathbf{x}_i)$ for any ($\mathbf{x}_i$,1) with a *positive* label in D_{train}

⇒ Maximize $P(0 \mid \mathbf{x}_i)$ for any ($\mathbf{x}_i$,0) with a *negative* label in D_{train}

Since $y_i \in \{0,1\}$ we can rewrite this to:

$$(\mathbf{w}^w, b^*) = \text{argmax}_{(\mathbf{w},b)} \prod_{(\mathbf{x}_i,y_i)\in D_{train}} P(1 \mid \mathbf{x}_i)^{y_i} \cdot [1 - P(1 \mid \mathbf{x}_i)]^{1-y_i}$$

For $y_i = 1$, this comes out to: $P(1 \mid \mathbf{x}_i)^1(1 - P(1 \mid \mathbf{x}_i))^0 = P(1 \mid \mathbf{x}_i)$

For $y_i = 0$, this is: $P(1 \mid \mathbf{x}_i)^0(1 - P(1 \mid \mathbf{x}_i))^1 = 1 - P(1 \mid \mathbf{x}_i) = P(0 \mid \mathbf{x}_i)$

Learning = Optimization = Loss Minimization

Learning = parameter estimation = optimization:

Given a particular class of model (logistic regression, Naive Bayes, …) and data D_{train}, find **the *best* parameters** for this class of model on D_{train}

If the model is a probabilistic classifier, think of optimization as Maximum Likelihood Estimation (**MLE**)

*"Best" = return (among all possible parameters for models of this class) parameters that assign the **largest probability** to* D_{train}

In general (incl. for probabilistic classifiers), think of optimization as **Loss Minimization**:

*"Best" = return (among all possible parameters for models of this class) parameters that have the **smallest loss** on* D_{train}

"**Loss**": how bad are the predictions of a model?

*The **loss function** we use to measure loss depends on the class of model*

$L(\hat{y}, y)$: how bad is it to predict $\hat{y}$ if the correct label is y ?

Conditional MLE $\Longrightarrow$ Cross-Entropy Loss

Conditional MLE: *Maximize probability* of labels in D_{train}

$$(\mathbf{w}^*, b^*) = \text{argmax}_{(\mathbf{w},b)} \prod_{(\mathbf{x}_i,y_i)\in D_{train}} P(y_i \mid \mathbf{x}_i)$$

$\Rightarrow$ Maximize $P(1 \mid \mathbf{x}_i)$ for any ($\mathbf{x}_i$,1) with a *positive* label in D_{train}

$\Rightarrow$ Maximize $P(0 \mid \mathbf{x}_i)$ for any ($\mathbf{x}_i$,0) with a *negative* label in D_{train}

Equivalently: *Minimize negative log prob.* of correct labels in D_{train}

$P(y_i \mid \mathbf{x}) = 0 \Leftrightarrow -\log(P(y_i \mid \mathbf{x})) = +\infty$ if y_i is the correct label for $\mathbf{x}$, this is the worst possible model

$P(y_i \mid \mathbf{x}) = 1 \Leftrightarrow -\log(P(y_i \mid \mathbf{x})) = 0$ if y_i is the correct label for $\mathbf{x}$, this is the best possible model

The negative log probability of the correct label is a **loss** function:

$-\log(P(y_i \mid \mathbf{x}_i))$ is **smallest** (0) when we assign **all** probability to the **correct** label

$-\log(P(y_i \mid \mathbf{x}_i))$ is **largest** ($+\infty$) when we assign **all** probability to the **wrong** label

This negative log likelihood loss is also called **cross-entropy loss**

From **loss** to per-example **cost**

Let's define the "**cost**" of our classifier on the whole dataset as its **average loss** on each of the *m* training examples:

$$\text{Cost}_{CE}(D_{\text{train}}) = \frac{1}{m} \sum_{i=1..m} -\log P(y_i \mid \mathbf{x}_i)$$

For each example:

$$-\log P(y_i \mid \mathbf{x}_i)$$

$$= -\log(\ P(1 \mid \mathbf{x}_i)^{y_i} \cdot P(0 \mid \mathbf{x}_i)^{1-y_i}\ )$$

[either $y_i = 1$ or $y_i = 0$]

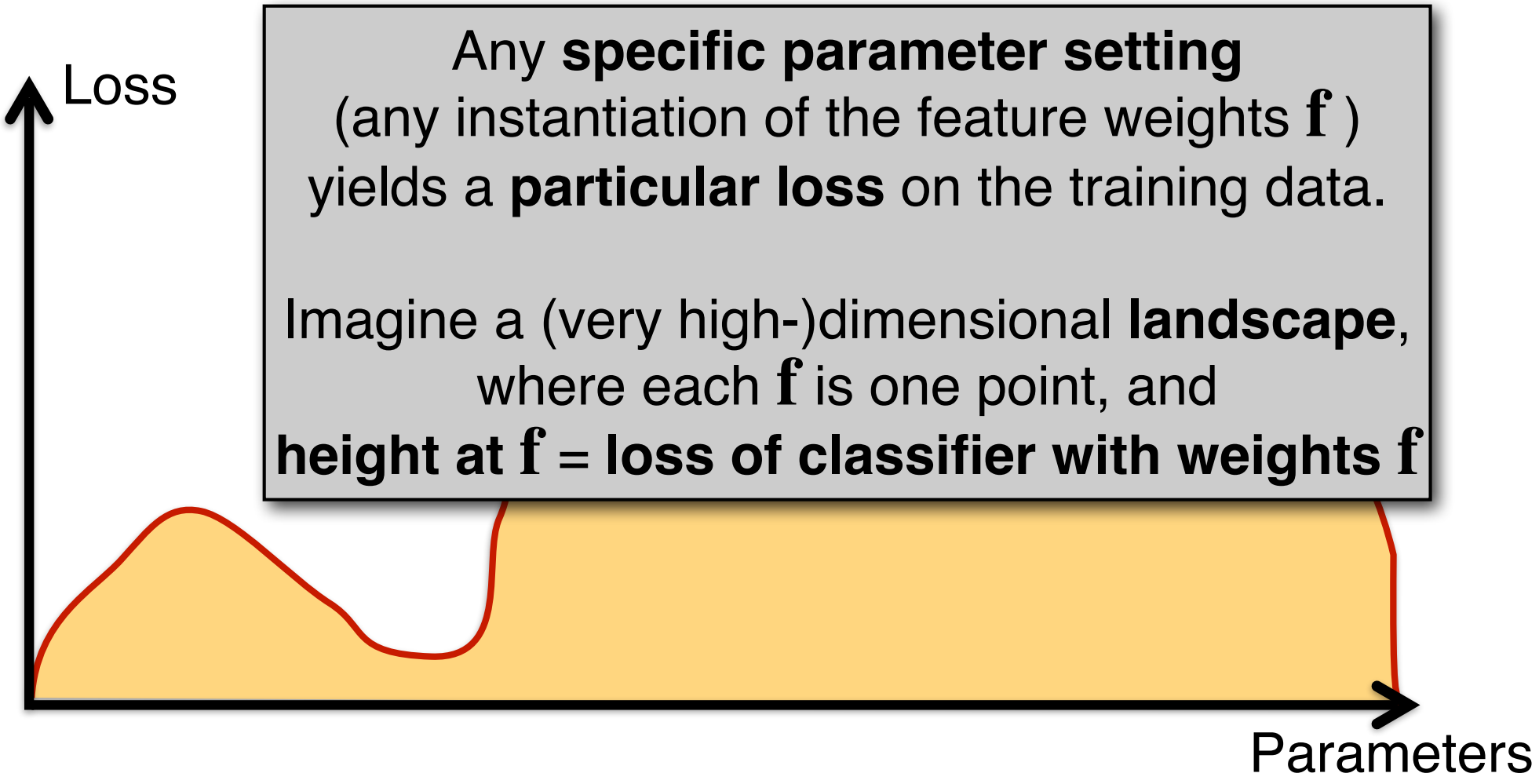
$$= -[\ y_i \log(\ P(1 \mid \mathbf{x}_i)) + (1 - y_i)\log(P(0 \mid \mathbf{x}_i))]$$

[moving the log inside]

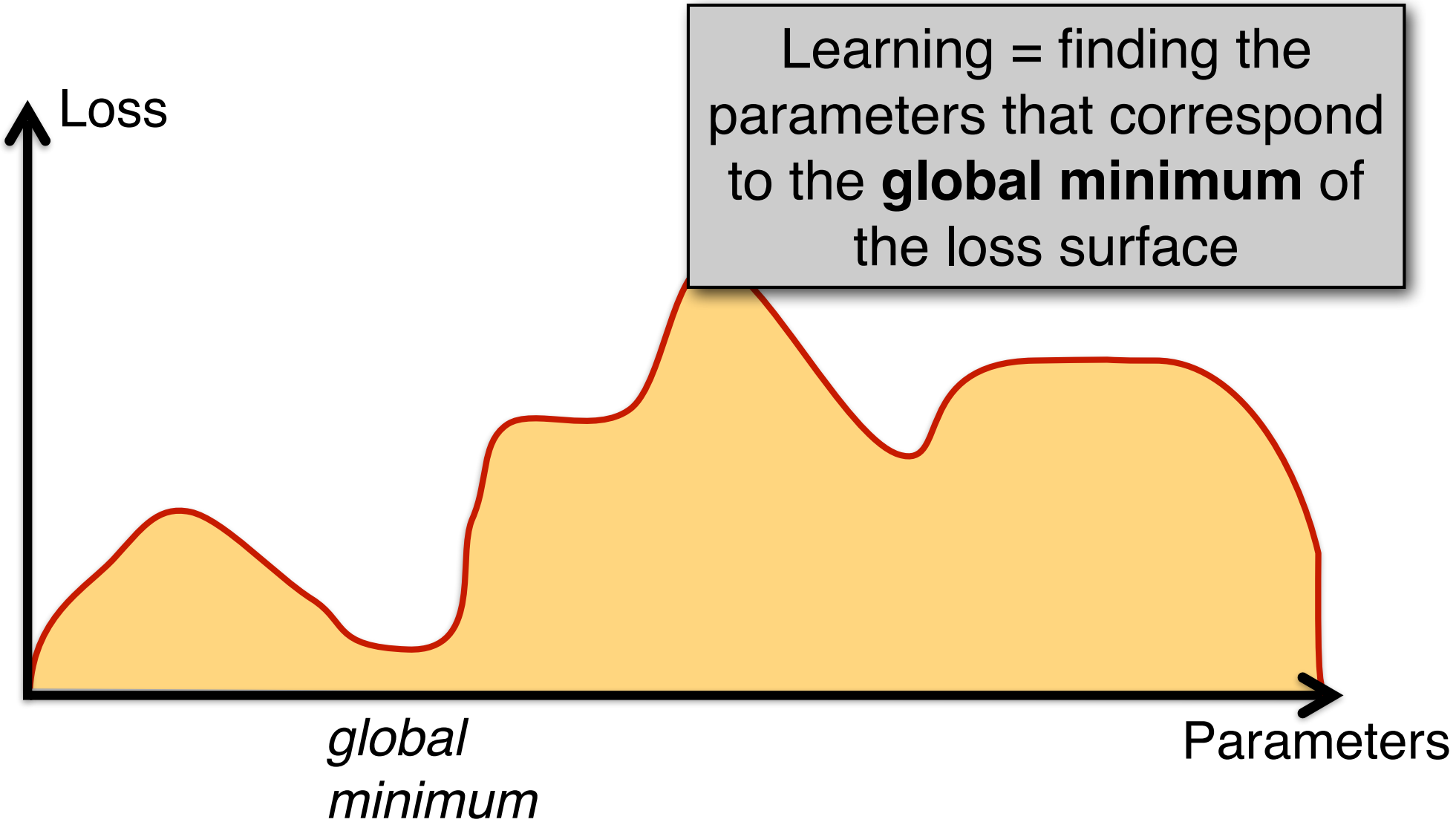
$$= -[\ y_i \log(\sigma(\mathbf{w}\mathbf{x}_i + b)) + (1 - y_i)\log(1 - \sigma(\mathbf{w}\mathbf{x}_i + b))]$$

[plugging in definition of $P(1 \mid \mathbf{x}_i)$]

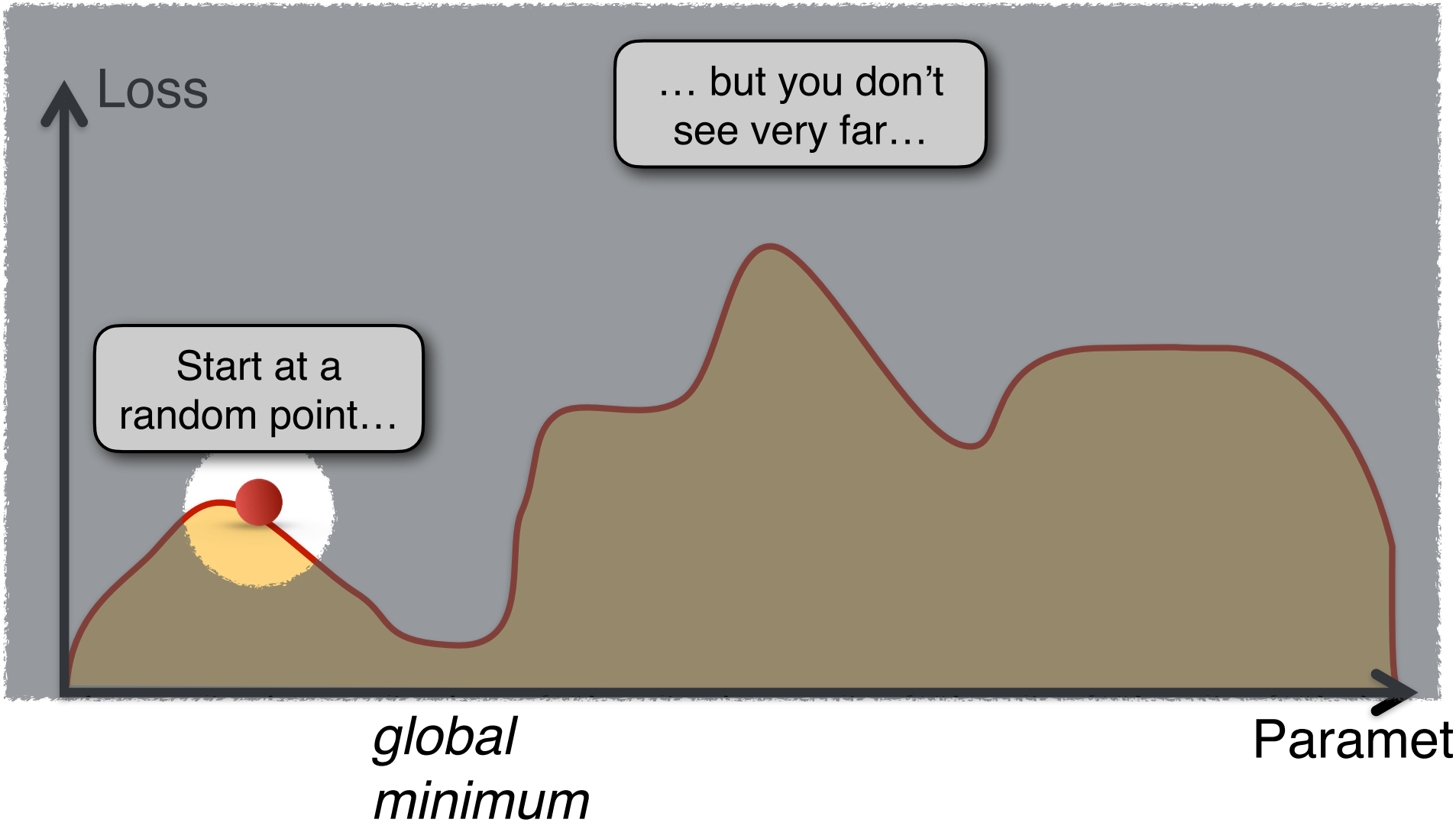
The loss surface



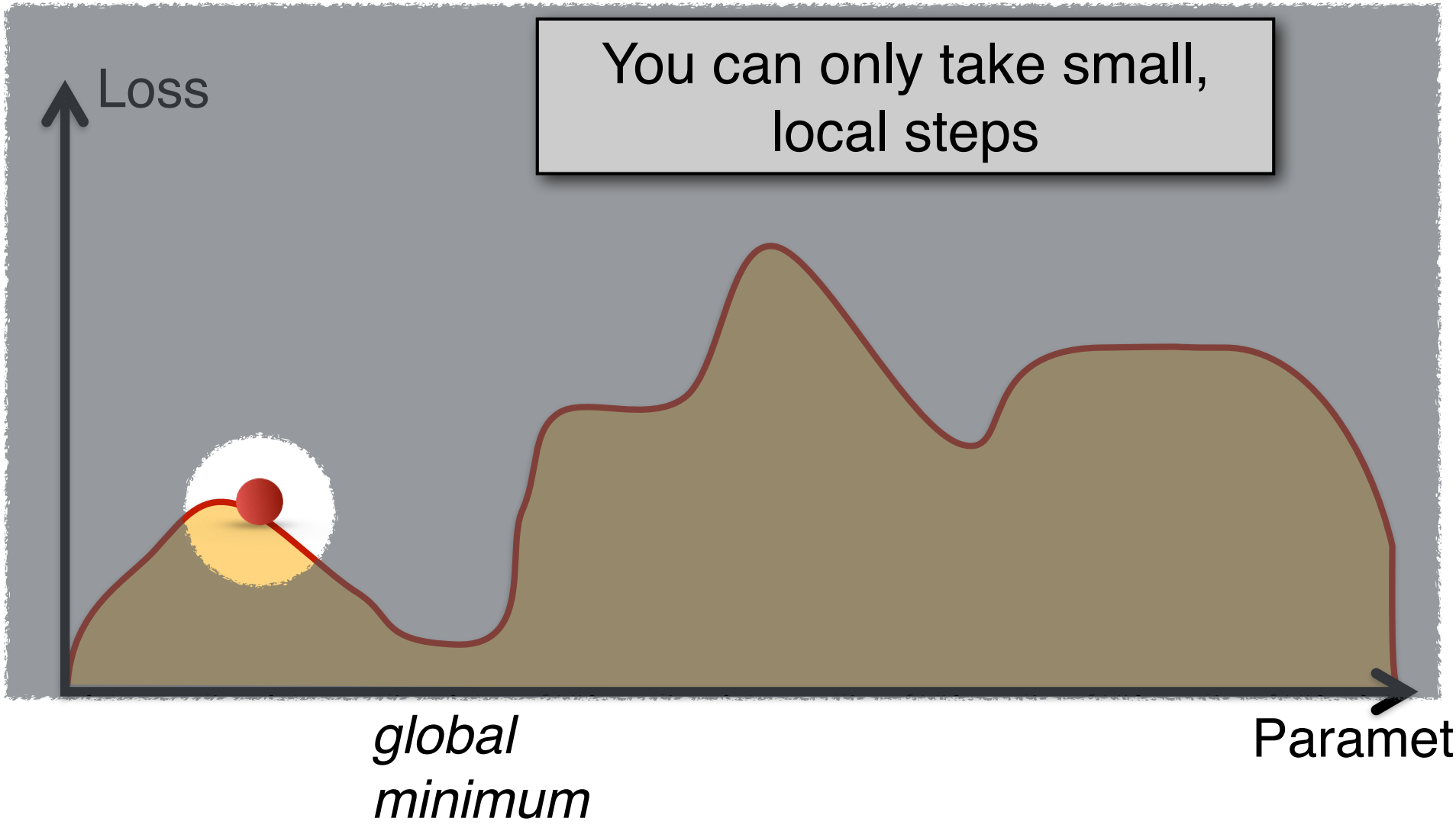
Learning = Moving in this landscape



Learning = Moving in this landscape



Learning = Moving in this landscape



Moving with Gradient Descent

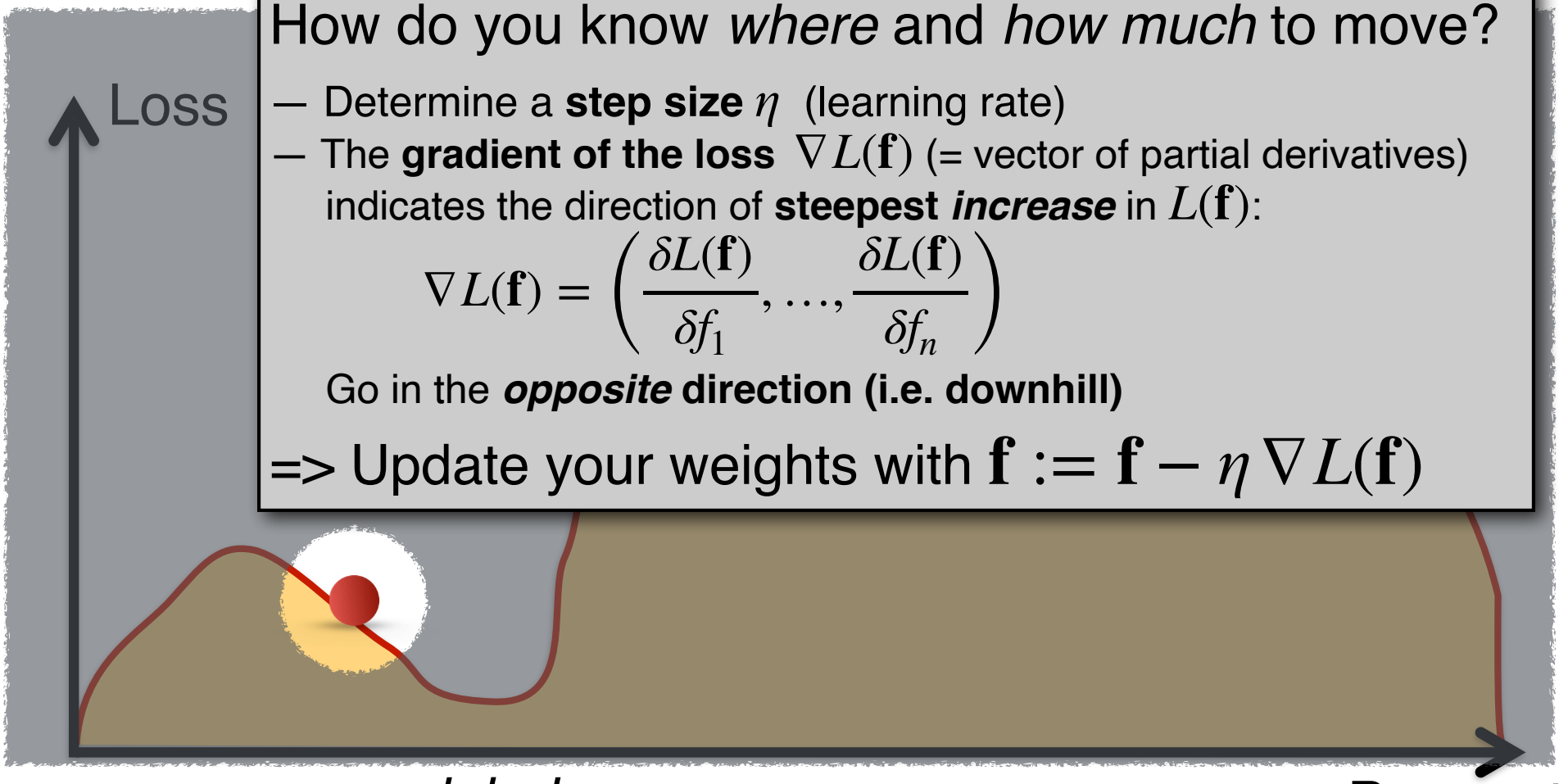
How do you know *where* and *how much* to move?

- Determine a **step size** η (learning rate)
- The **gradient of the loss** $\nabla L(\mathbf{f})$ (= vector of partial derivatives) indicates the direction of **steepest *increase*** in $L(\mathbf{f})$:

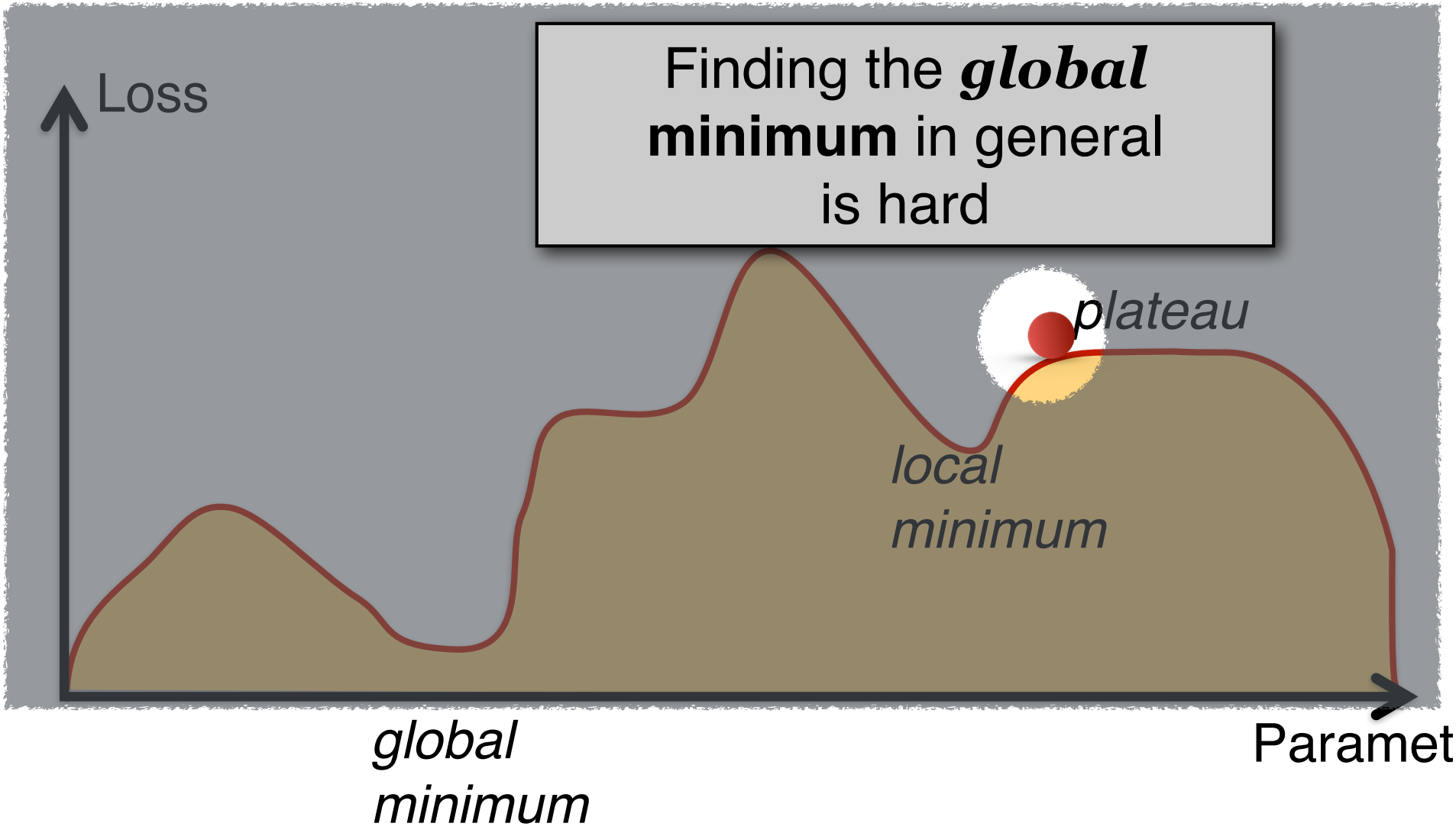
$$\nabla L(\mathbf{f}) = \left(\frac{\delta L(\mathbf{f})}{\delta f_1}, \ldots, \frac{\delta L(\mathbf{f})}{\delta f_n} \right)$$

Go in the ***opposite*** **direction (i.e. downhill)**

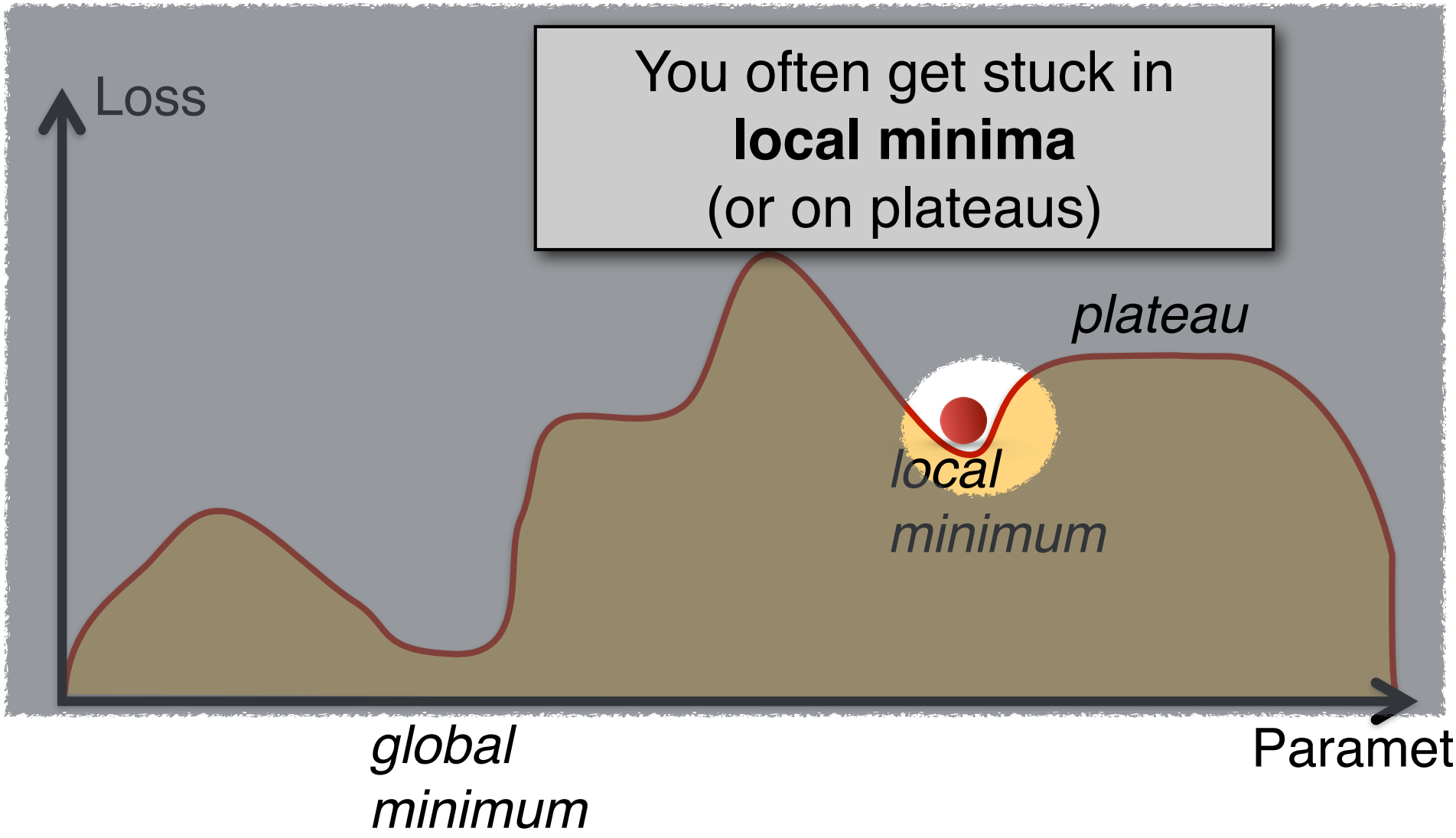
=> Update your weights with $\mathbf{f} := \mathbf{f} - \eta \nabla L(\mathbf{f})$



Gradient Descent finds *local* optima



Gradient Descent finds *local* optima



(Stochastic) Gradient Descent

— We want to find parameters that have minimal cost (loss) on our training data.

— But we don't know the whole loss surface.

— However, the gradient of the cost (loss) of our current parameters tells us how the slope of the loss surface at the point given by our current parameters

— And then we can take a (small) step in the right (downhill) direction (to update our parameters)

Gradient descent:
Compute loss for entire dataset before updating weights

Stochastic gradient descent:
Compute loss for one (randomly sampled) training example before updating weights

Stochastic Gradient Descent

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

Gradient for Logistic Regression

Computing the gradient of the loss for example $\mathbf{x}_i$ and weight $\mathbf{w}_j$ is very simple (x_{ji}: j-th feature of $\mathbf{x}_i$)

$$\frac{\delta L(\mathbf{w}, b)}{\delta w_j} = [\sigma(\mathbf{w}\mathbf{x}_i + b) - y_i]x_{ji}$$

More details

The **learning rate** η affects **convergence**

- There are many options for setting the **learning rate**: fixed, decaying (as a function of time), adaptive,…
- Often people use more complex schemes and optimizers

Mini-batch training computes the gradient on a small batch of training examples at a time.

- Often more stable than SGD.

Regularization keeps the size of the weights under control

- L1 or L2 regularization


The End