CS447: Natural Language Processing

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

Lecture 6: Logistic Regression continued, Intro to Neural Nets

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Where are we at?

Language Models: $P(w_1 \ldots. w_N)$

— N-Gram models

Classification for NLP: $P(c \mid w_1 \ldots w_n)$

— Naive Bayes

— Logistic Regression (⇐ to be wrapped up today)

Today: Introduction to neural networks

— From logistic regression to classification with neural nets

— A simple neural n-gram model

Future lectures:

— From words to vectors (distributional similarity, embeddings)

— Recurrent nets (getting rid of the n-gram assumption)

Logistic Regression

Probabilistic classifiers

A probabilistic classifier returns the *most likely* class *y* for input **x**:

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

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

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 **x** with *P*(**x**|*y*)

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.

P(Y | **X**) with Logistic Regression

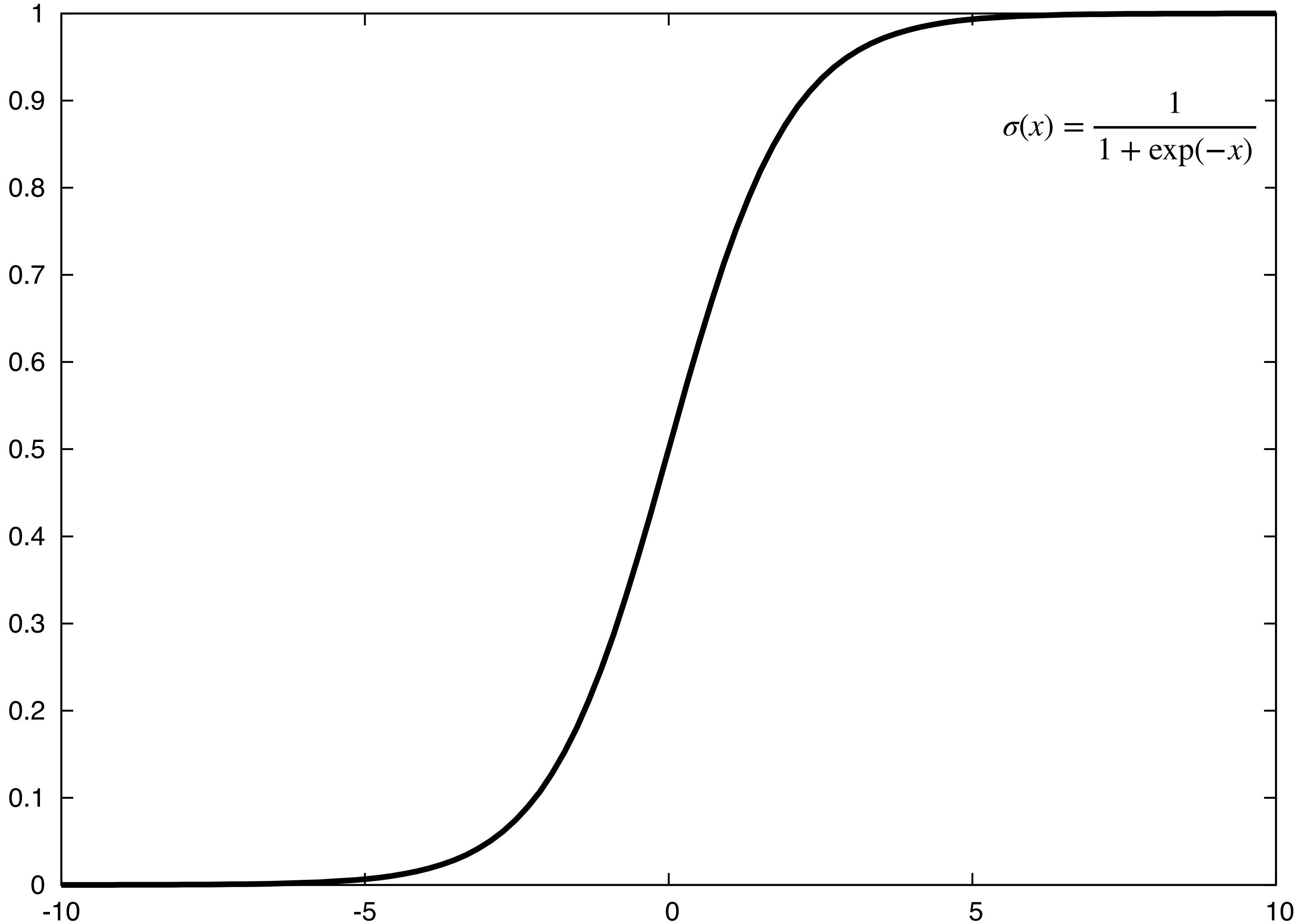
Task: Model $P(y \mid \mathbf{x})$ for any input (feature) vector **x**=(x_1,…,x_n)
Idea: Learn feature weights **w**=(w_1,…,w_n) (and a bias term *b*) to capture how important each feature x_i is for predicting the class *y*

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 **w** and one bias term *b*

The sigmoid function



Using Logistic Regression

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

1) **Design**:
Decide how to map raw inputs to feature vectors **x**
[Advantage for discriminative models *P*(y |x): features do not need to be independent]

2) **Training**:
Learn parameters **w** and *b* on training da

3) **Testing**:
Use the classifier to classify unseen inputs

Feature Design

Features capture **properties** of the input

- Does the input contain a particular unigram, bigram, longer phrase…? (Or: what's the frequency of a particular unigram, bigram, phrase in the input?)
- Is a word capitalized? Does it end in a particular substring? Does it contain a particular character?
- Features may also be computed by other classifiers

We typically specify **feature templates** and then use **feature selection** (or **regularization**) methods to automatically identify useful **features** (instantiations of these features)

- Feature Template:
  "any pair of adjacent words that appears >2x in training data"
- Actual features: "an apple",…., "zillion zebras"

Learning parameters **w** and *b*

Training objective:
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}

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 ⟹ Cross-Entropy Loss

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

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

⇒ 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}

Equivalently: *Minimize negative log prob.* of 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 largest ($+\infty$) when we assign all probability to the wrong label,

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

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

Training with Cross-Entropy Loss

Binary classification:
The training examples $(\mathbf{x}_i, y_i)$ have gold labels $y_i \in \{0,1\}$

A logistic regression classifier, defined by parameters $(\mathbf{w}, b)$, computes the conditional probability of a label given the input as

$$P(Y_i = 1 \mid \mathbf{x}_i) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$
$$P(Y_i = 0 \mid \mathbf{x}_i) = 1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

Define the loss of this classifier on any training example $(\mathbf{x}_i, y_i)$ as the negative log probability $-\log P_{\mathbf{w},b}(Y_i = y_i \mid \mathbf{x}_i)$ that it assigns to that example.

$$\mathrm{L}_{\mathbf{w},b}(\mathbf{x}_i, y_i) = -\log P_{\mathbf{w},b}(y_i \mid \mathbf{x}_i)$$

Training objective: find parameters $(\mathbf{w}, b)$ that minimize this loss

From loss to per-example cost

Let's define the "cost" of our classifier on the whole dataset as the average loss of 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:

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

From loss to per-example cost

Let's define the "cost" of our classifier on the whole dataset as the average loss of 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 $y_i = 0$]

From loss to per-example cost

Let's define the "cost" of our classifier on the whole dataset as the average loss of 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]

From loss to per-example cost

Let's define the "cost" of our classifier on the whole dataset as the average loss of 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 $y_i = 0$]

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

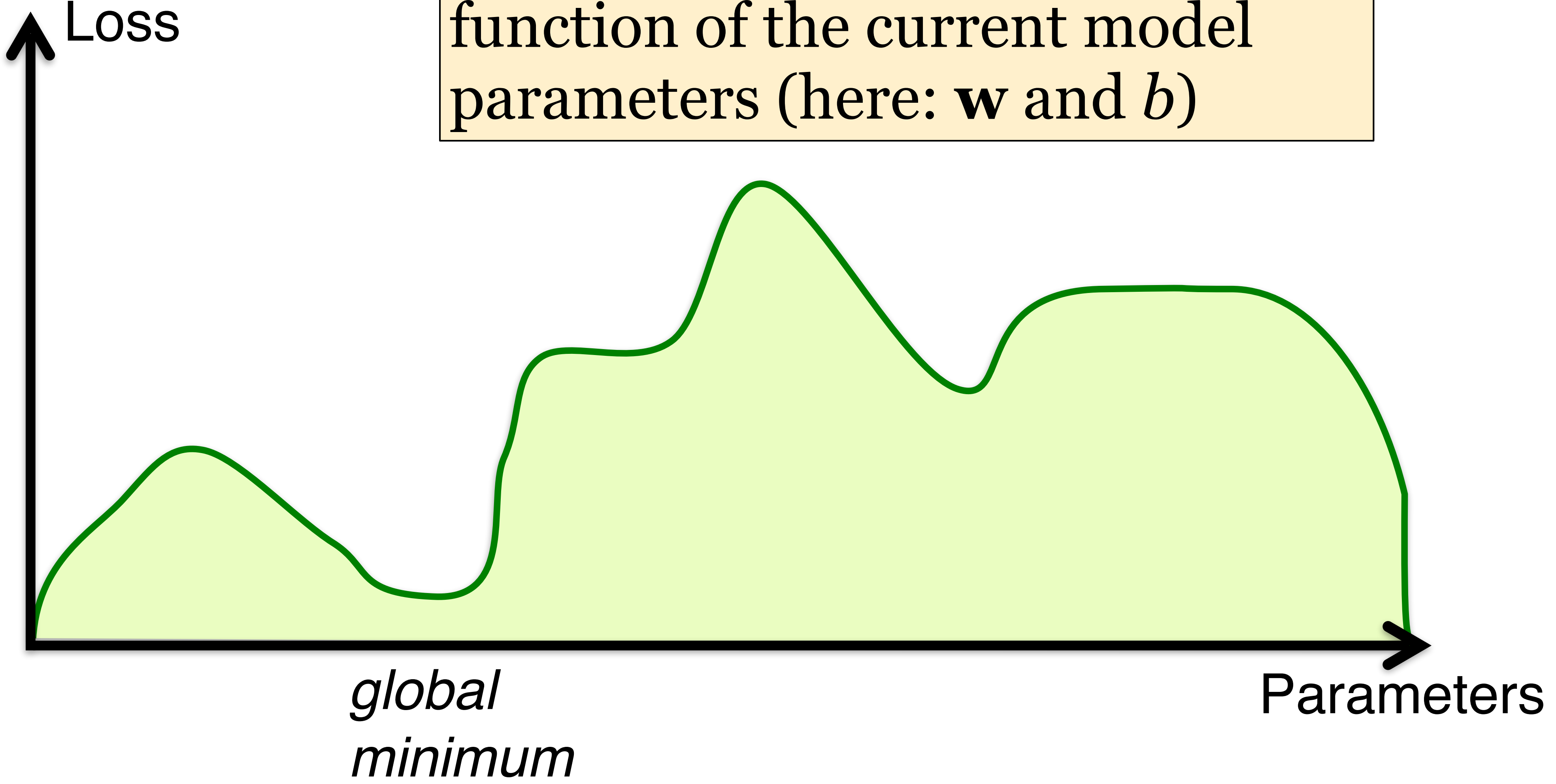
[moving the log inside]

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

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

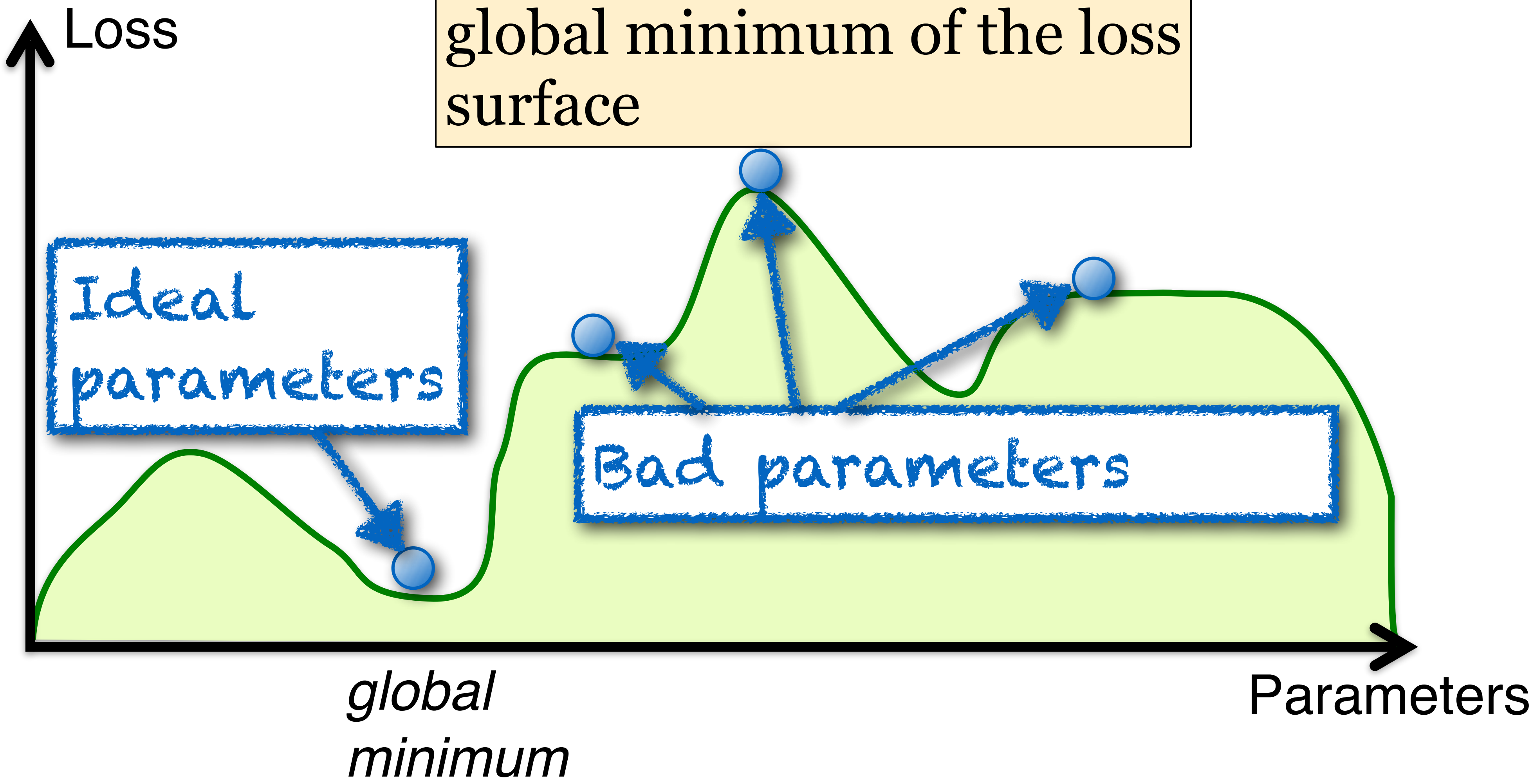
The loss surface

The loss is a (highly non-convex) function of the current model parameters (here: **w** and *b*)



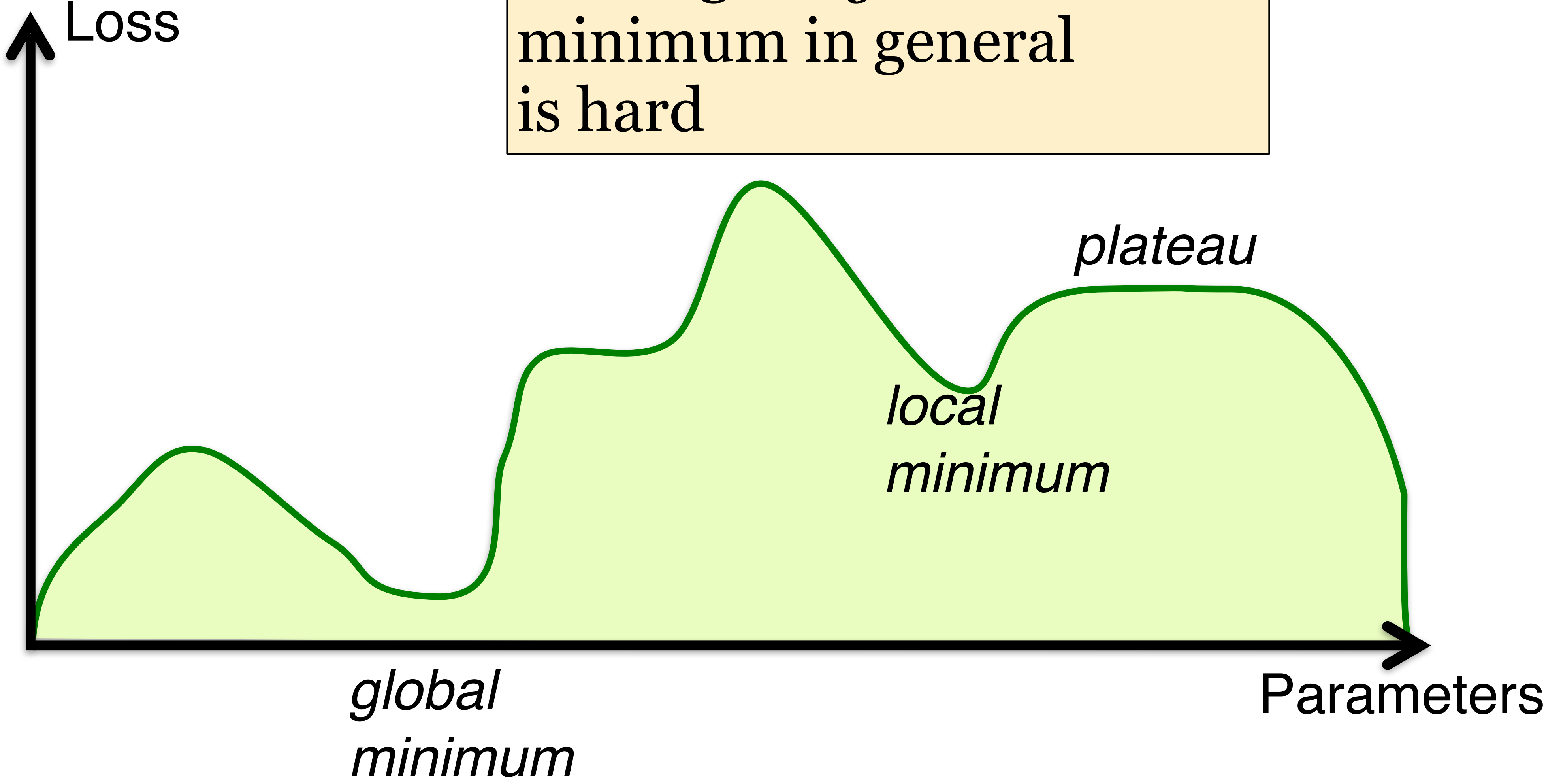
The loss surface

Learning = finding the global minimum of the loss surface



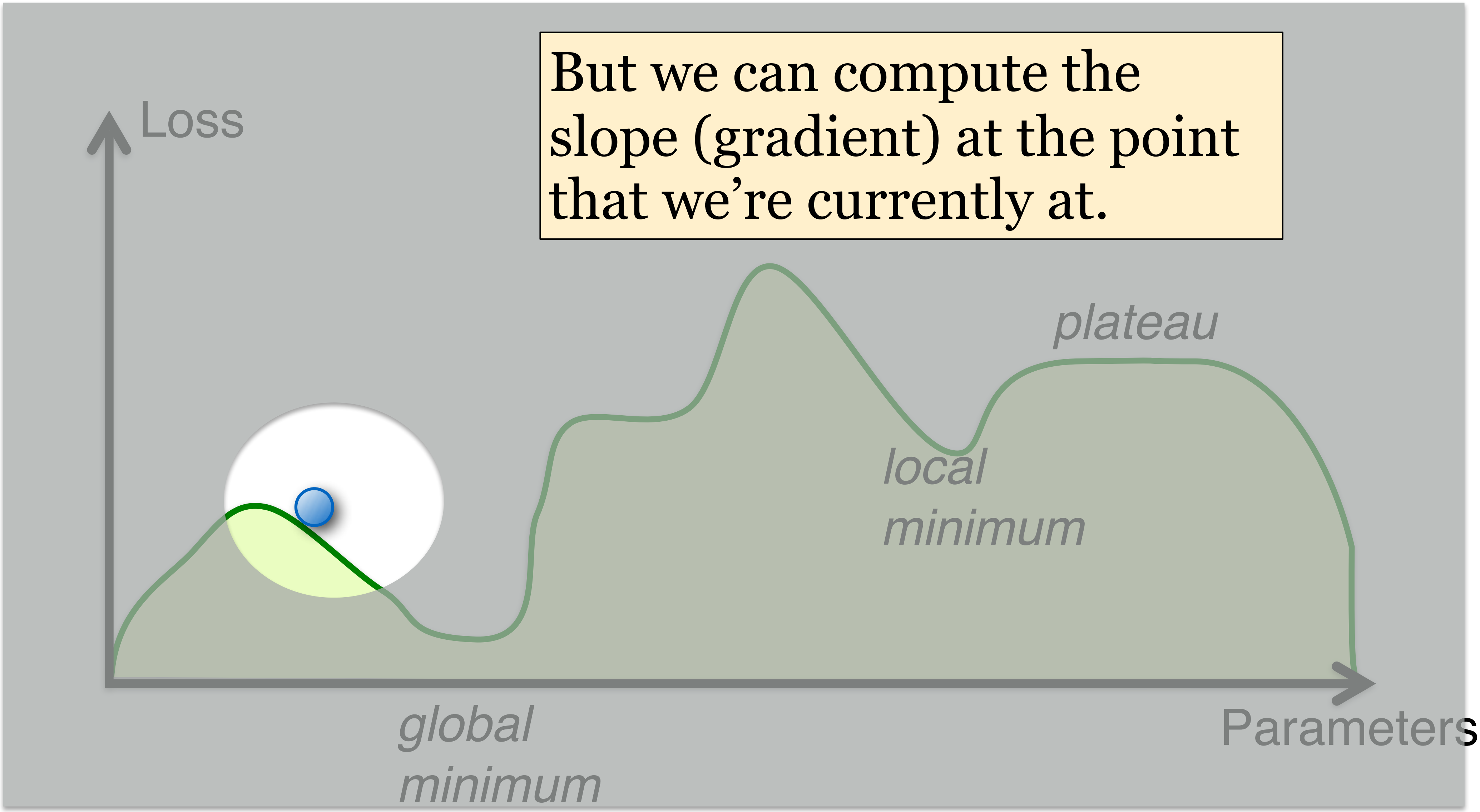
The loss surface

Finding the *global* minimum in general is hard



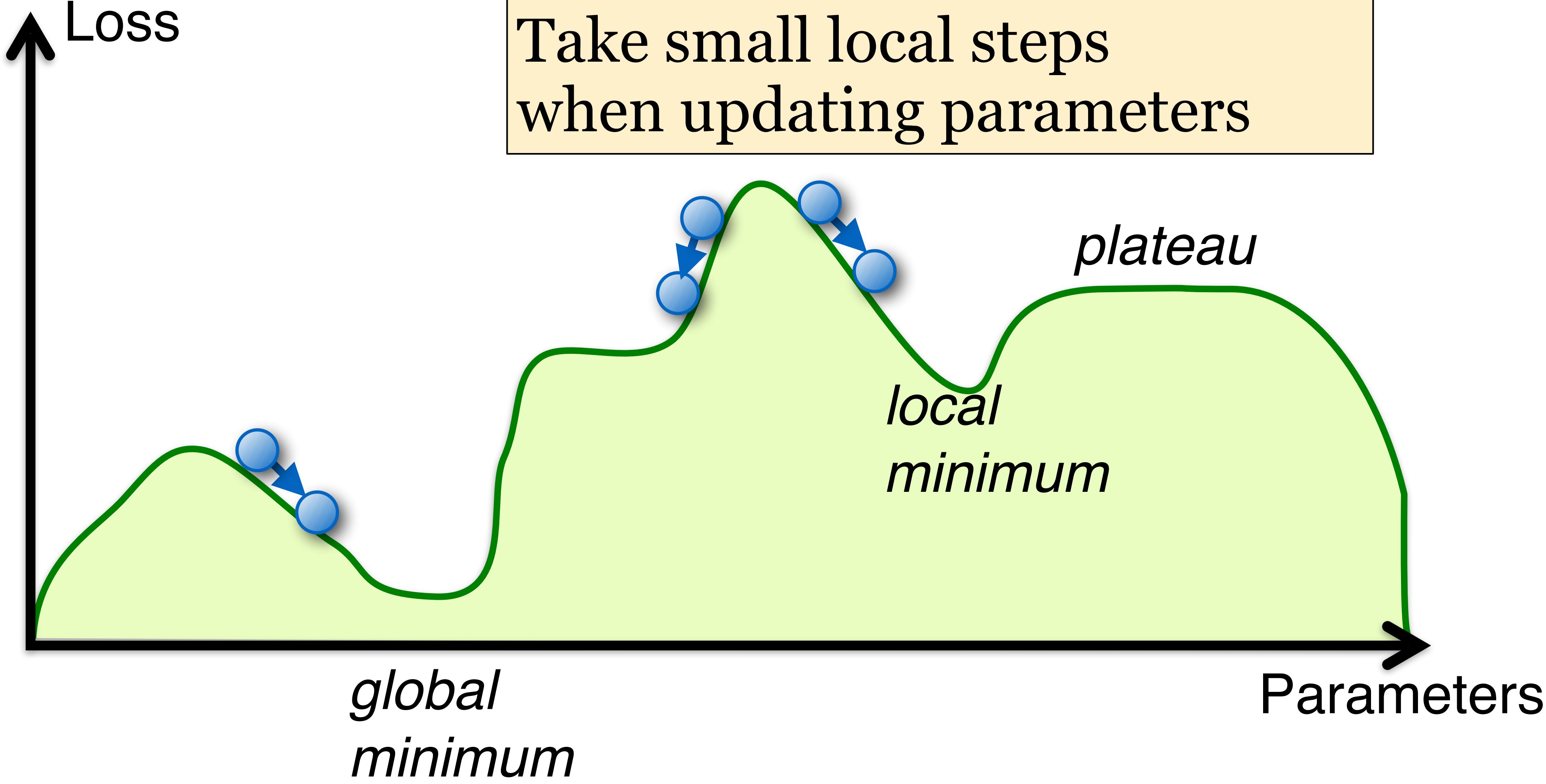
Gradient of the loss

Gradient of the loss



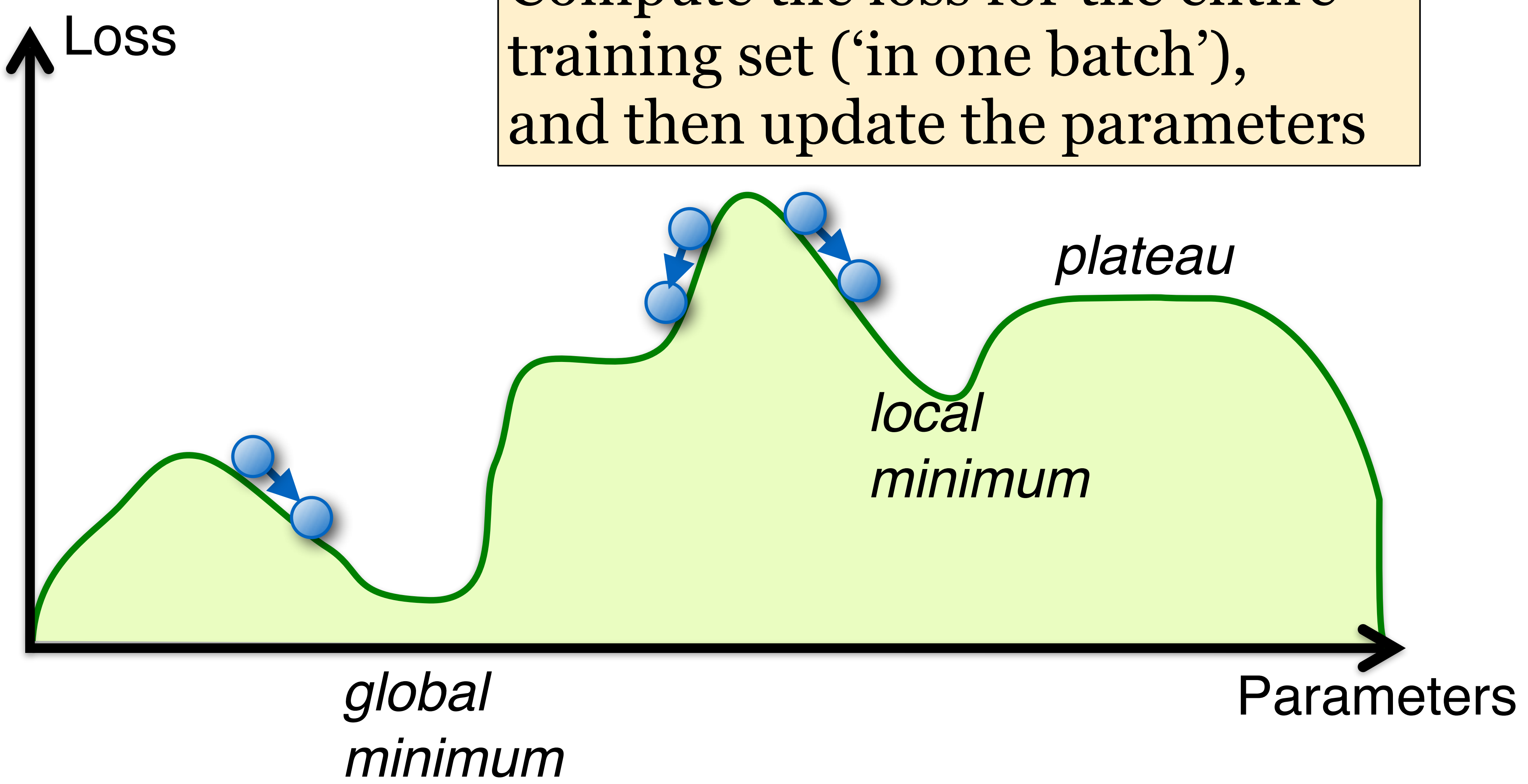
Gradient descent

Basic idea:
Take small local steps
when updating parameters



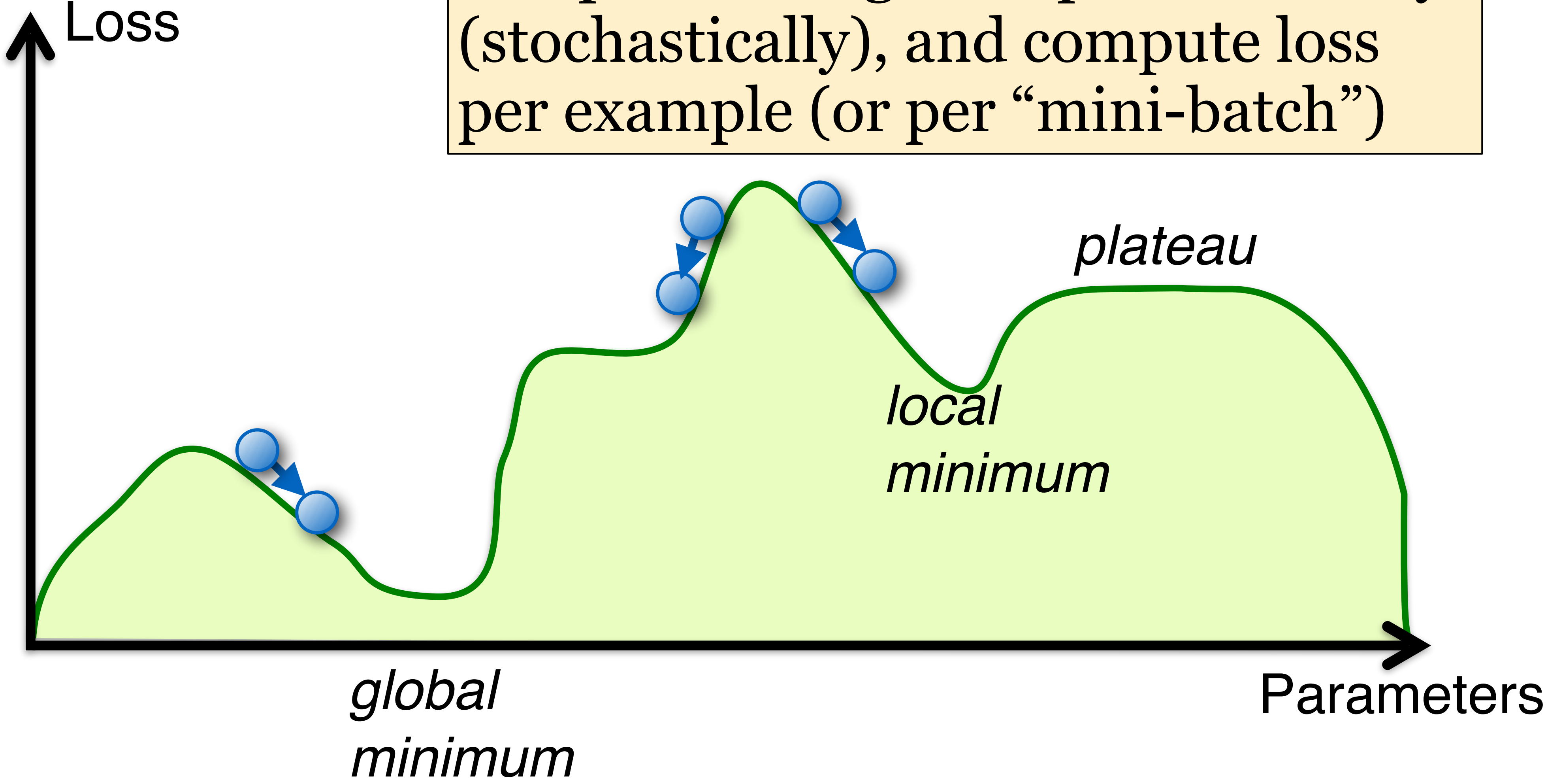
Batch Gradient descent

Compute the loss for the entire training set ('in one batch'), and then update the parameters



Stochastic Gradient descent

Sample training examples randomly (stochastically), and compute loss per example (or per "mini-batch")



(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

P(Y | **X**) with Logistic Regression

Task: Model $P(y \mid \mathbf{x})$ for any input (feature) vector **x**=(x_1,…,x_n)
Idea: Learn feature weights **w**=(w_1,…,w_n) (and a bias term *b*) to capture how important each feature x_i is for predicting the class *y*

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 **w** and one bias term *b*

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

$$P(Y=y_i \mid \mathbf{x}) = \text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} = \frac{\exp(-(\mathbf{w}_i\mathbf{x} + b_i))}{\sum_{j=1}^{K} \exp(-(\mathbf{w}_j\mathbf{x} + b_j))}$$

Parameters to learn: one feature weight vector **w** and one bias term *b* ***per class.***

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 $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$
    #   y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$

$\theta \leftarrow 0$
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 $\mathbf{x}_i$)

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

Multiclass classification

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 **w** and one bias term *b*

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

$$P(Y=y_i \mid \mathbf{x}) = \text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{K}\exp(z_j)} = \frac{\exp(-(\mathbf{w}_i\mathbf{x}+b_i))}{\sum_{j=1}^{K}\exp(-(\mathbf{w}_j\mathbf{x}+b_j))}$$

Parameters to learn: one feature weight vector **w** and one bias term *b* ***per class.***

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

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

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

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

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

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

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

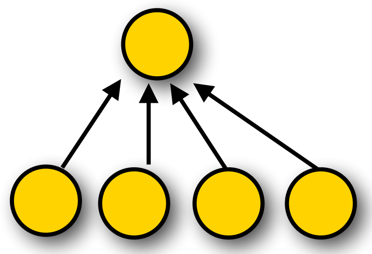
➜ Think of binary lr. as multinomial lr. with $\exp(-(\mathbf{w}_1\mathbf{x} + b_1)) = 1$ (i.e. where $\mathbf{w}_1$ is set to the null vector and $b_1 := 0$)

From logistic regression to neural nets

What are neural nets?

Simplest variant: single-layer feedforward net

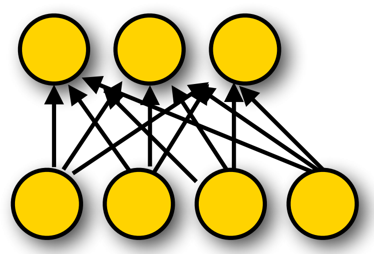
For binary classification tasks:
Single output unit
Return 1 if y > 0.5
Return 0 otherwise



Output unit: scalar *y*

Input layer: vector **x**

For multiclass classification tasks:
K output units (a vector)
Each output unit
y_i = class i
Return $\text{argmax}_i(y_i)$



Output layer: vector **y**

Input layer: vector **x**

Multiclass models: $\text{softmax}(y_i)$

Multiclass classification = predict one of K classes.

Return the class i with the highest score: $\text{argmax}_i(y_i)$

In neural networks, this is typically done by using the **softmax** function, which maps real-valued vectors in R^N into a distribution over the N outputs

For a vector $\mathbf{z} = (z_0 \ldots z_K)$: $P(i) = \text{softmax}(z_i) = \exp(z_i) \;/\; \sum_{k=0..K} \exp(z_k)$

This is just logistic regression

Single-layer feedforward networks

Single-layer (linear) feedforward network

$y = \mathbf{wx} + b$ (binary classification)

w is a weight vector, *b* is a bias term (a scalar)

This is just a linear classifier (aka Perceptron)

(the output *y* is a linear function of the input **x**)

Single-layer non-linear feedforward networks:

Pass $\mathbf{wx} + b$ through a non-linear activation function,

e.g. $y = \tanh(\mathbf{wx} + b)$

Nonlinear activation functions

Sigmoid (logistic function): $\sigma(x) = 1/(1 + e^{-x})$

Useful for output units (probabilities) [0,1] range

Hyperbolic tangent: $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$

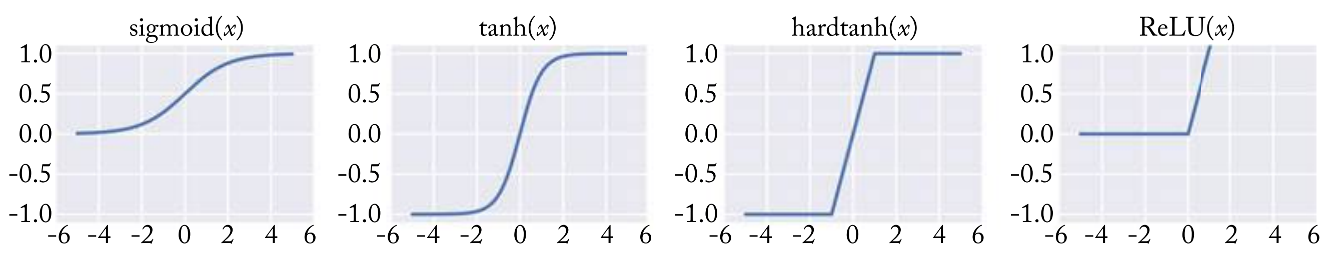
Useful for internal units: [-1,1] range

Hard tanh (approximates tanh)

$\text{htanh}(x) = -1 \text{ for } x < -1,\ 1 \text{ for } x > 1,\ x \text{ otherwise}$

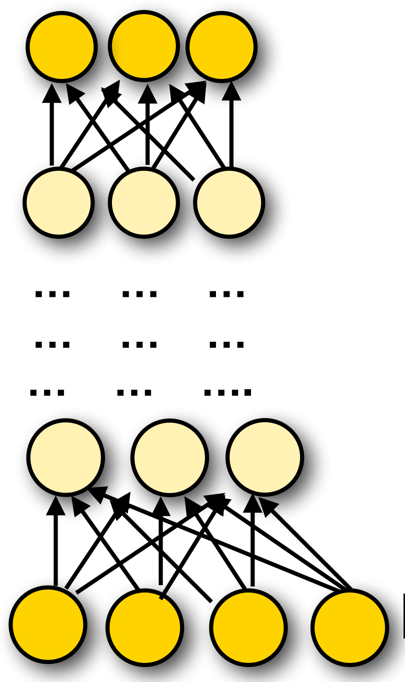
Rectified Linear Unit: $\text{ReLU}(x) = \max(0, x)$

Useful for internal units



Multi-layer feedforward networks

We can generalize this to multi-layer feedforward nets



Output layer: vector $\mathbf{y}$

Hidden layer: vector $\mathbf{h}_n$

Hidden layer: vector $\mathbf{h}_1$

Input layer: vector $\mathbf{x}$

Neural Language Models

What is a language model?

Probability distribution over the strings in a language, typically factored into distributions $P(w_i \mid \ldots)$ for each word:

$$P(\mathbf{w}) = P(w_1 \ldots w_n) = \prod_i P(w_i \mid w_1 \ldots w_{i-1})$$

N-gram models assume each word depends only preceding n−1 words:

$$P(w_i \mid w_1 \ldots w_{i-1}) =_{def} P(w_i \mid w_{i-n+1} \ldots w_{i-1})$$

To handle variable length strings, we assume each string starts with n−1 start-of-sentence symbols (BOS), or 〈S〉
and ends in a special end-of-sentence symbol (EOS) or 〈\S〉

An n-gram model $P(w \mid w_1 \ldots w_k)$ as a feedforward net (naively)

— The **vocabulary** V contains *n* types (incl. UNK, BOS, EOS)
— We want to condition each word on *k* preceding words

— [Naive] Each **input word** $w_i \in V$ (that we're conditioning on) is an ***n*-dimensional one-hot vector** $v(w) = (0,\ldots 0, 1, 0 \ldots 0)$
— Our **input layer** $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ has $n{\times}k$ elements
— To predict the probability over output words, the **output layer** is a softmax over n elements

$$P(w \mid w_1 \ldots w_k) = \mathrm{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

With (say) one hidden layer $\mathbf{h}$ we'll need two sets of parameters, one for **h** and one for the output

Naive neural n-gram model

Architecture:

Input Layer: $\mathbf{x} = [v(w_1) \ldots . v(w_k)]$

$v(w) = E_{[w]}$

Hidden Layer: $\mathbf{h} = g(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)$

Output Layer: $P(w \mid w1 \ldots wk) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$

Parameters:

Weight matrices and biases:

first layer: $\mathbf{W}^1 \in R^{k \cdot \dim(V) \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in R^{\dim(\mathbf{h})}$

second layer: $\mathbf{W}^2 \in R^{\dim(\mathbf{h}) \times |V|}$ $\mathbf{b}^2 \in R^{|V|}$

Naive neural n-gram model

Advantage over non-neural n-gram model:

— The hidden layer captures interactions among context words

— Increasing the order of the n-gram requires only a small linear increase in the number of parameters.

dim($\mathbf{W}^1$) goes from $k \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$ to $(k+1) \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$

— Increasing the vocabulary also leads only to a linear increase in the number of parameters

But: with a one-hot encoding and dim(V) ≈ 10K or so, this model still requires a LOT of parameters to learn.

#parameters going to hidden layer: $k \cdot \dim(V) \cdot \dim(\mathbf{h})$,

with $\dim(\mathbf{h}) = 300$, $\dim(V) = 10{,}000$ and $k{=}3$: 9,000,000

Plus #parameters going to output layer: $\dim(\mathbf{h}) \cdot \dim(V)$

with $\dim(\mathbf{h}) = 300$, $\dim(V) = 10{,}000$: 3,000,000

Neural n-gram models

Advantages over traditional n-gram models:

- Increasing the order requires only a small linear increase in the number of parameters.

  $\mathbf{W}^1$ goes from $R^{k \cdot \dim(emb) \times \dim(\mathbf{h})}$ to $R^{(k+1) \cdot \dim(emb) \times \dim(\mathbf{h})}$
- Increasing the number of words in the vocabulary also leads only to a linear increase in the number of parameters
- Easy to incorporate more context: just add more input units
- Easy to generalize across contexts (embeddings!)

Neural n-gram models

Naive neural language models have similar shortcomings to standard n-gram models

- Models get very large (and sparse) as n increases
- We can't generalize across similar contexts
- Markov (independence) assumptions in n-gram models are too strict

Solutions offered by less naive neural models:

- Do not represent context words as distinct, discrete symbols (i.e. very high-dimensional one-hot vectors), but use a dense low-dimensional vector representation where similar words have similar vectors [next class]
- Use recurrent nets that can encode variable-lengths contexts [later class]