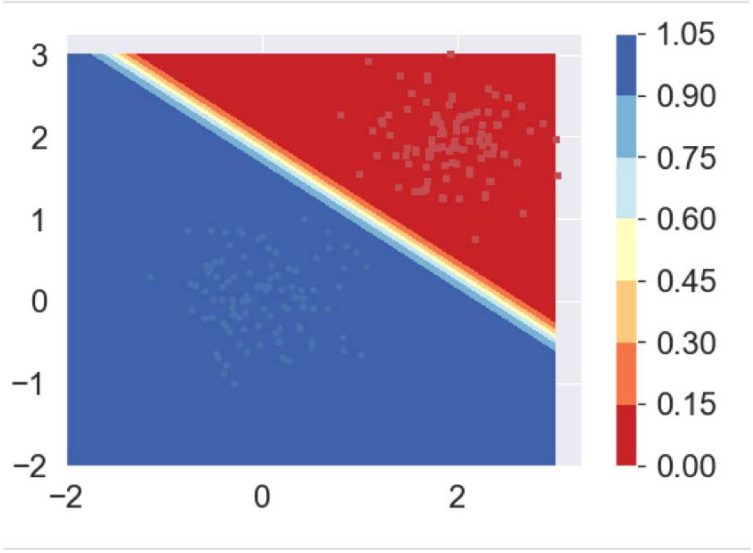
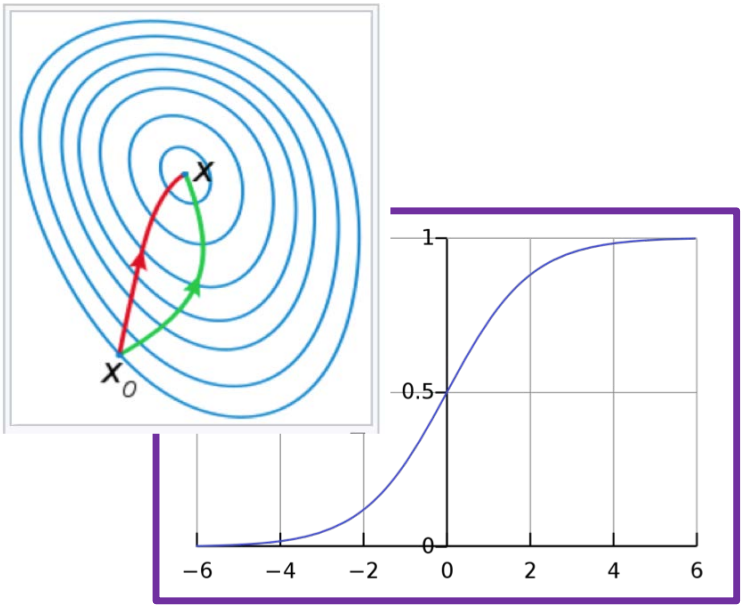
Tufts COMP 135: Introduction to Machine Learning

https://www.cs.tufts.edu/comp/135/2019s/

Logistic Regression 2/2





Many slides attributable to:
Erik Sudderth (UCI)
Finale Doshi-Velez (Harvard)
James, Witten, Hastie, Tibshirani (ISL/ESL books)

Prof. Mike Hughes

Logistics

- HW3 due tonight
  - Please annotate pages in Gradescope!
  - Remember: Turn in on time!
- Project 1 out later tonight
- No recitation next week (President's day)

Objectives Today: Logistic Regression Unit 2/2

Concept Check-in "Quiz"

Logistic Regression: A Probabilistic Classifier

- Computing log loss and its gradient
- Training via gradient descent
  - How to pick step size
  - Advanced: Line search, Using 2nd Derivatives
  - How to scale to big data: stochastic gradient descent

Check-in Q1:

When training Logistic Regression, we minimize the log loss on the training set.

$$\text{log\_loss}(y, \hat{p}) = -y \log \hat{p} - (1 - y) \log(1 - \hat{p})$$

$$\min_{w,b} \sum_{n=1}^{N} \text{log\_loss}(y_n, \hat{p}(x_n, w, b))$$

Can you provide 3 justifications for why this log loss objective is sensible?

Why is an L2 penalty useful?

Check-in Q2:

Consider the definition of the sigmoid function:

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

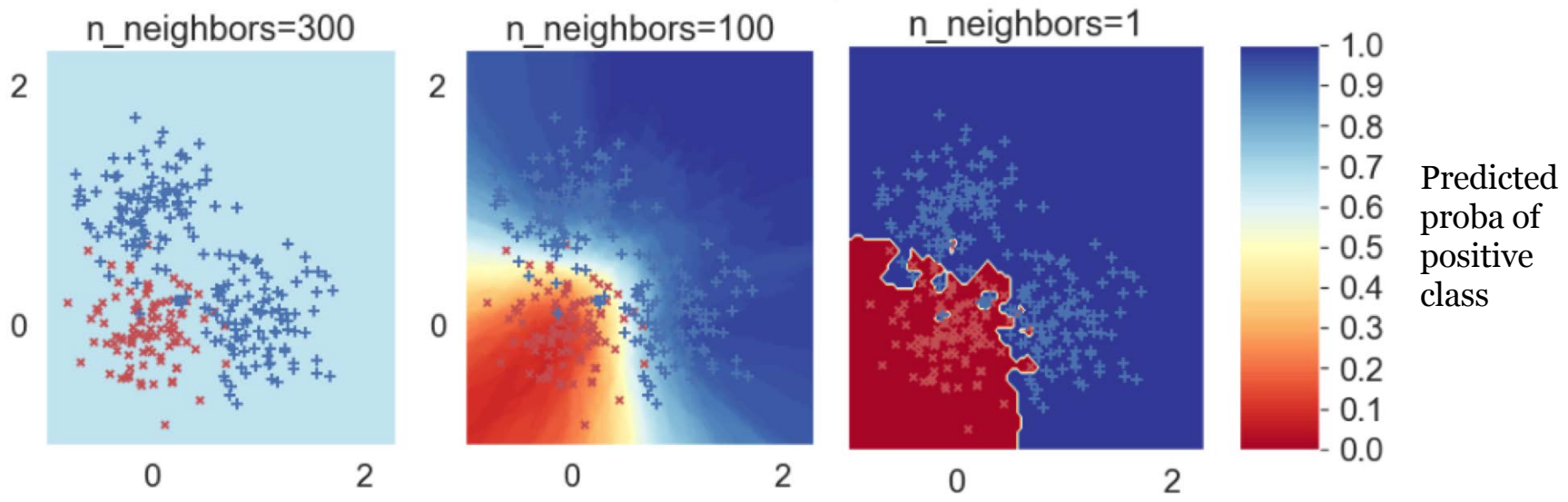
What could go wrong with this implementation?

```
def sigmoid(z):
    return np.exp(z) / (1 + np.exp(z))
```

How would we fix the issue?

Check-in Q3

K-Nearest Neighbors looks like a pretty accurate classifier…



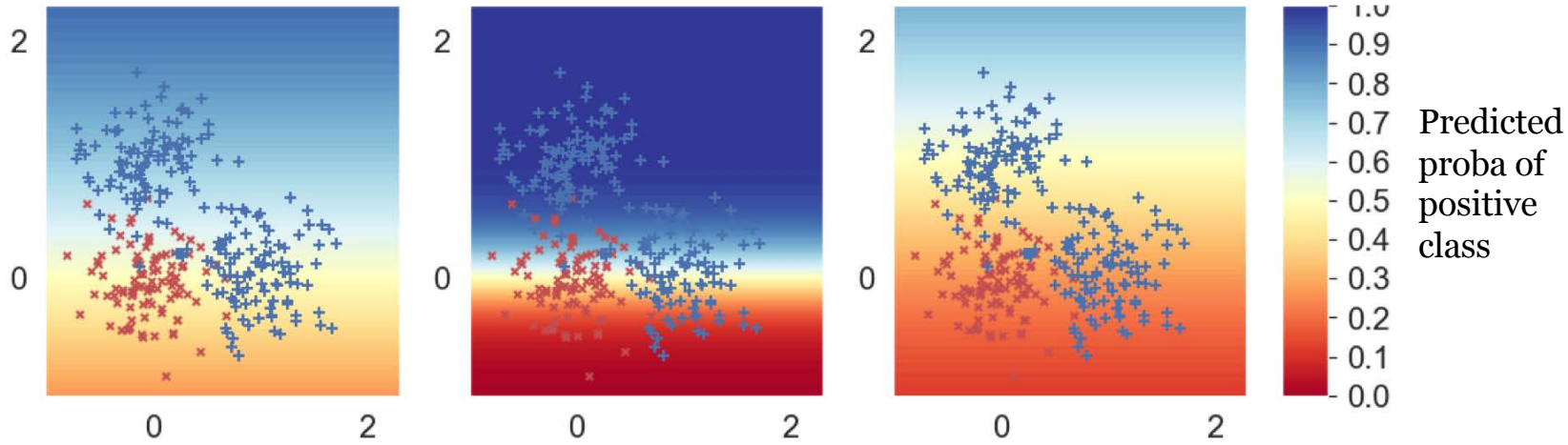
What are its primary drawbacks?

Check-in Q4

Consider logistic regression classifier for 2D features

What is the value (approximately)

of w_1, w_2, and bias for each plot below?

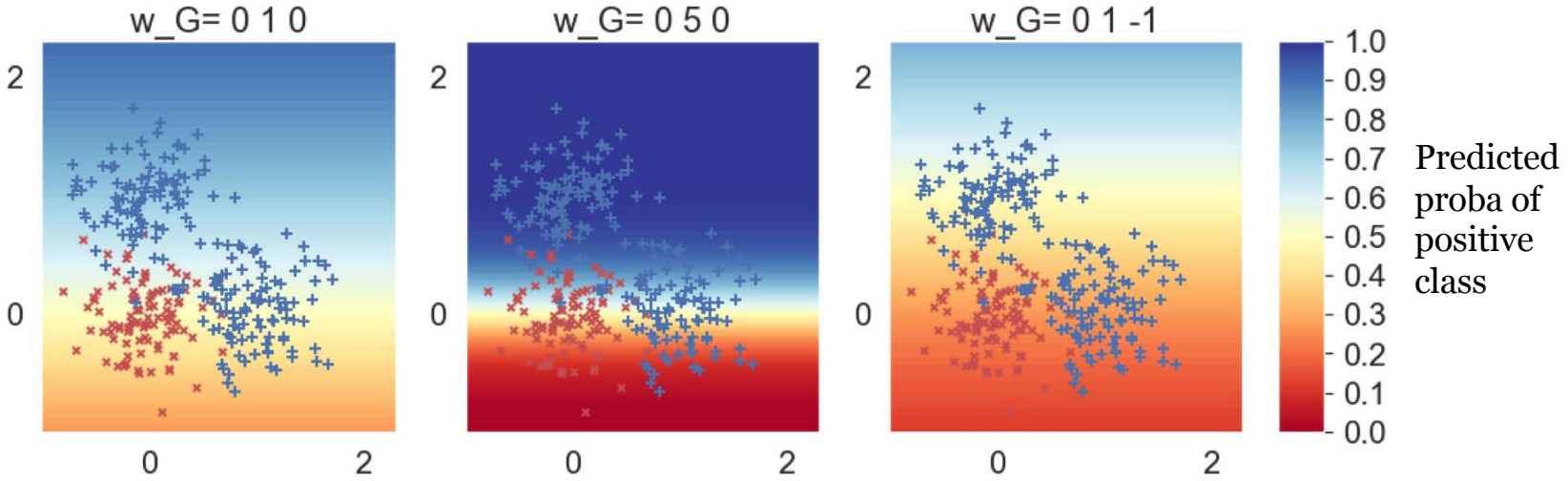


Check-in Q4

Consider logistic regression classifier for 2D features

What is the value (approximately)

of w_1, w_2, and bias for each plot below?

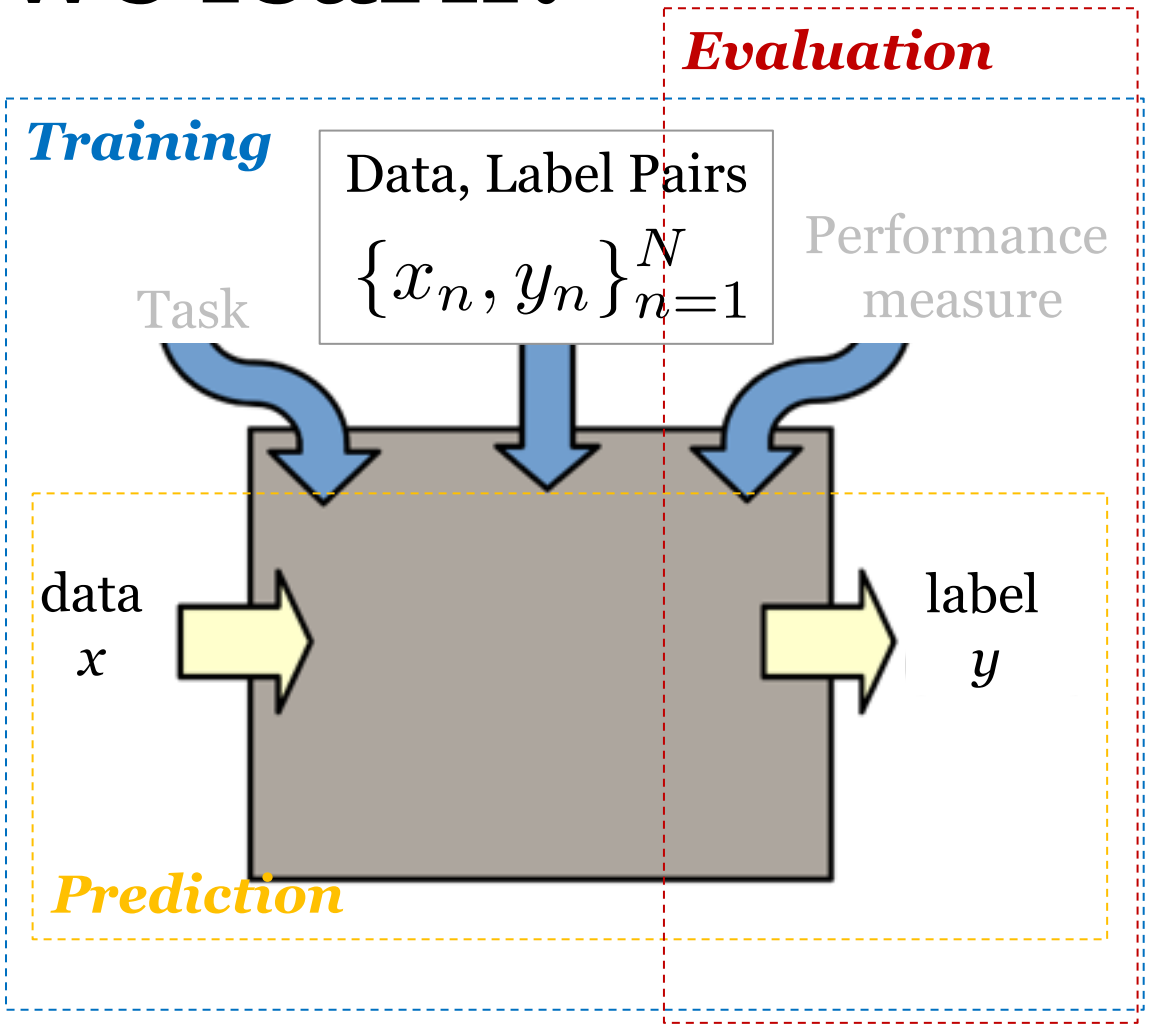


What will we learn?

Supervised Learning

Unsupervised Learning

Reinforcement Learning



Task: Binary Classification

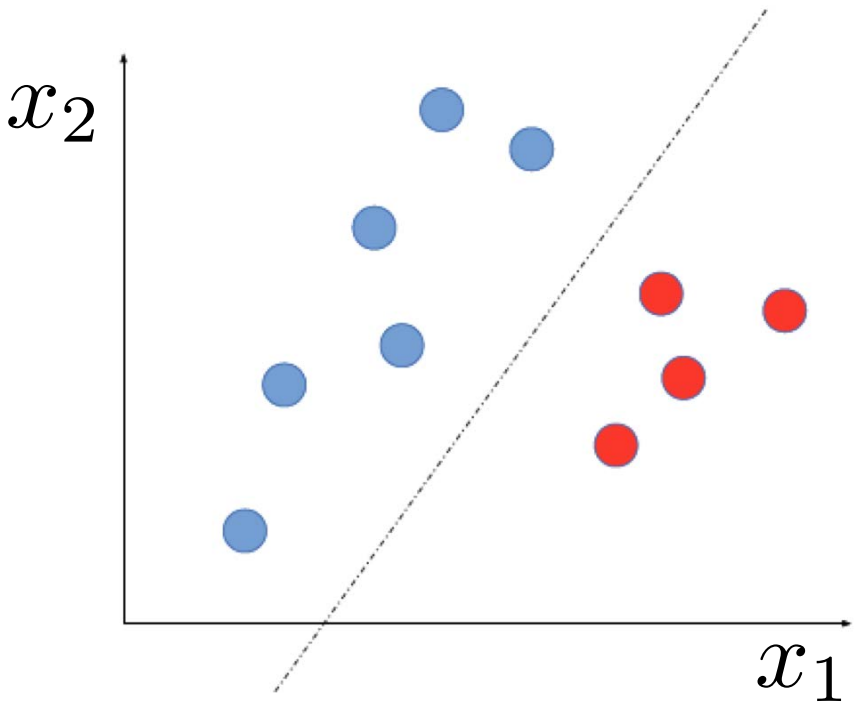
Supervised Learning

binary classification

Unsupervised Learning

Reinforcement Learning

y is a binary variable (red or blue)



Example: Hotdog or Not

https://www.theverge.com/tldr/2017/5/14/15639784/hbo-silicon-valley-not-hotdog-app-download

Binary Prediction

Goal: Predict label (0 or 1) given features x

- Input: $x_i \triangleq [x_{i1}, x_{i2}, \ldots x_{if} \ldots x_{iF}]$

  "features"
  "covariates"
  "attributes"

  Entries can be real-valued, or other numeric types (e.g. integer, binary)

- Output: $y_i \in \{0, 1\}$

  "responses" or "labels"

  Binary label (0 or 1)

```
>>> yhat_N = model.predict(x_NF)
>>> yhat_N[:5]
[0, 0, 1, 0, 1]
```

Probability Prediction

Goal: Predict probability of label given features

- Input: $x_i \triangleq [x_{i1}, x_{i2}, \ldots x_{if} \ldots x_{iF}]$

  "features"
  "covariates"
  "attributes"

  Entries can be real-valued, or other numeric types (e.g. integer, binary)

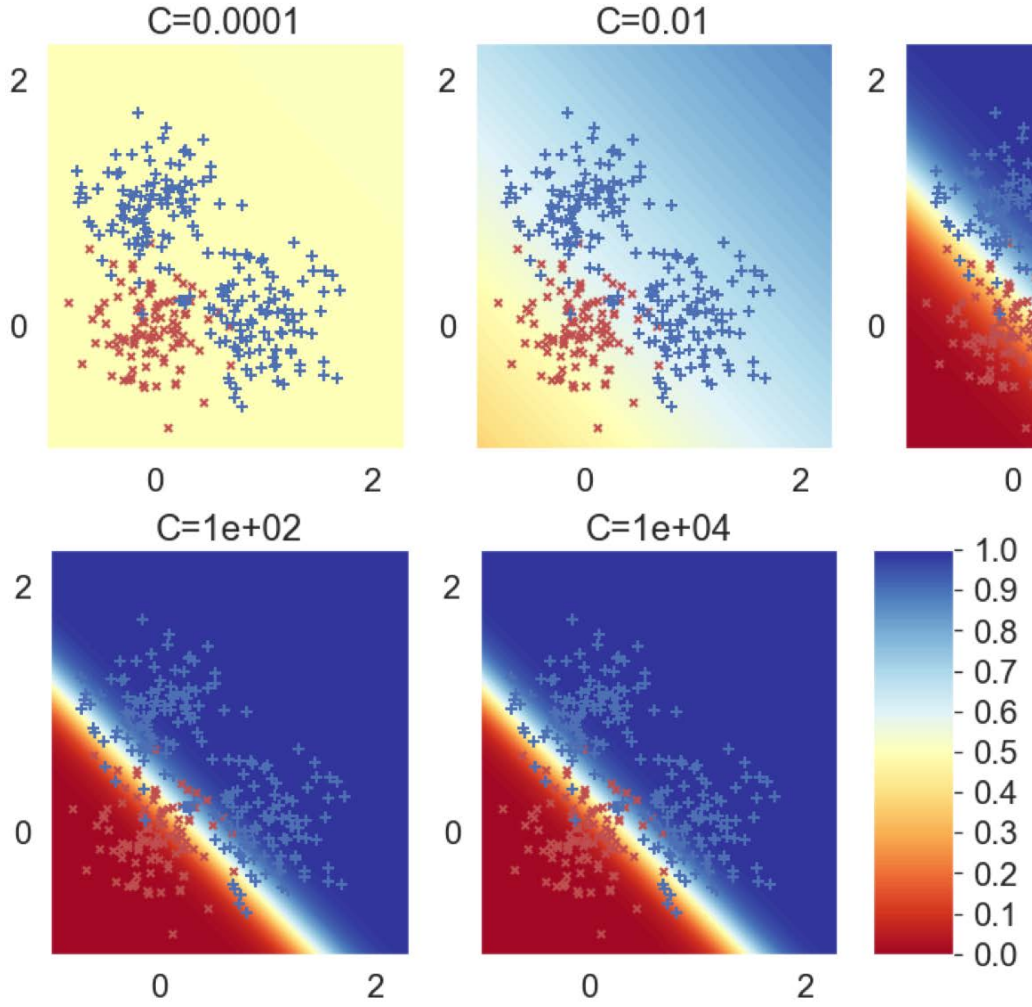
- Output: $\hat{p}_i \triangleq p(Y_i = 1 | x_i)$

  "probability"

  Value between 0 and 1
  e.g. 0.001, 0.513, 0.987

```
>>> yproba_N2 = model.predict_proba(x_NF)
>>> yproba1_N = model.predict_proba(x_NF)[:,1]
>>> yproba1_N[:5]
[0.143, 0.432, 0.523, 0.003, 0.994]
```

Logistic Regr: Predicted Probas



Function is monotonically increasing in one direction

***Decision boundaries** will be linear*

Computing gradients for Logistic Regression (LR)

Simplified LR notation

- Feature vector with first entry constant

$$\tilde{x}_n = [1 \; x_{n1} \; x_{n2} \; \ldots \; x_{nF}]$$

- Weight vector (first entry is the "bias")

$$w = [w_0 \;\; w_1 \;\; w_2 \ldots w_F]$$

- "Score" value z (real number, -inf to +inf)

$$z_n = w^T \tilde{x}_n$$

Simplifying the log likelihood

$$
\begin{aligned}
J(z_n) &\triangleq \log p(Y_n = y_n | z_n) \\
&= y_n \log \sigma(z_n) + (1 - y_n) \log(1 - \sigma(z_n)) \\
&= y_n \log \sigma(z_n) + (1 - y_n) \log \sigma(-z_n) \\
&= y_n \log \frac{e^{z_n}}{1 + e^{z_n}} + (1 - y_n) \log \frac{1}{1 + e^{z_n}} \\
&= y_n z_n - \log\left(1 + e^{z_n}\right)
\end{aligned}
$$

Gradient of the log likelihood

$$z_n = w^T \tilde{x}_n$$

Log likelihood

$$J(z_n(w)) = y_n z_n - \log(1 + e^{z_n})$$

Gradient w.r.t. weight on feature f

$$\frac{d}{dw_f} J(z_n(w)) = \frac{d}{dz_n} J(z_n) \cdot \frac{d}{dw_f} z(w)$$

Simplifying yields: $$= (y_n - \sigma(z_n))\, x_{nf}$$

Gradient descent for LR

Gradient descent for L2 penalized LR

$$\min_{\boldsymbol{w}} \; -\sum_i \log \; p(y_i \,|\boldsymbol{x}_i; \boldsymbol{w}) + \frac{\lambda}{2} \|\boldsymbol{w}\|_2^2$$

Start with $\boldsymbol{w}^0 = 0$, step size s

for iteration $t = 0, \ldots, (T-1)$

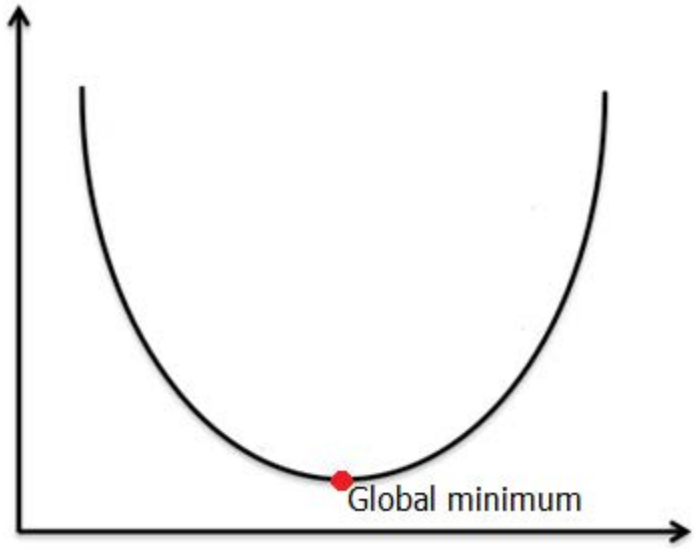
$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \mathrm{s}\, \nabla L(\boldsymbol{w}^t)$

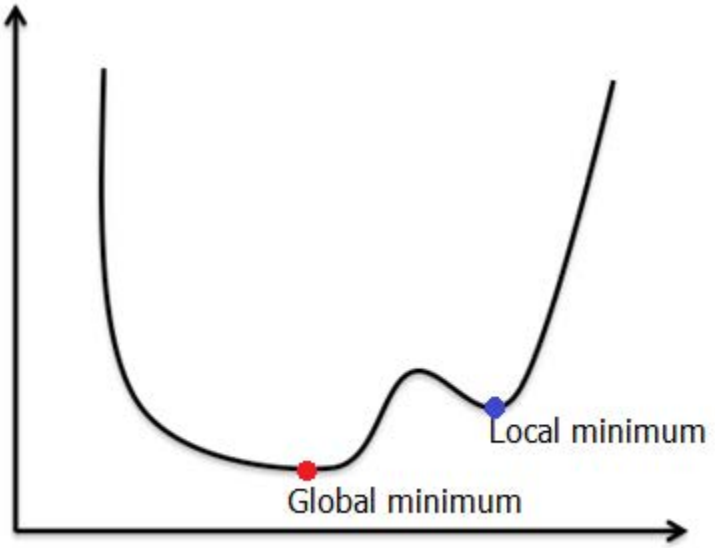
if $L(\boldsymbol{w}^{t+1}) - L(\boldsymbol{w}^t) < \delta$

break

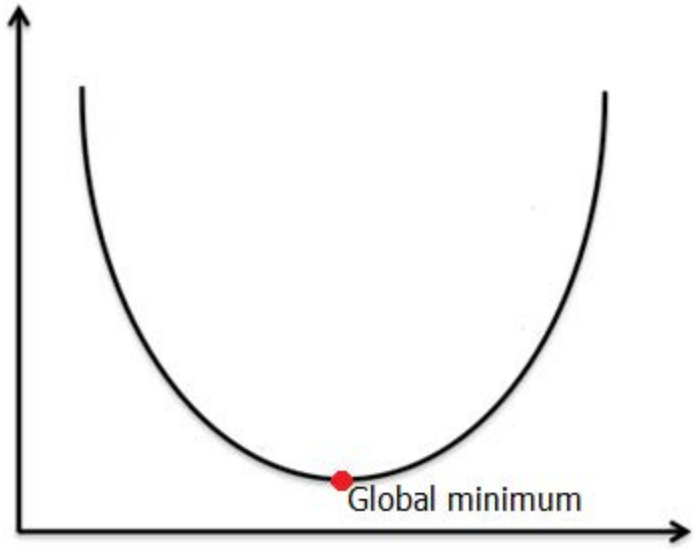
return $\boldsymbol{w}^T$

Will gradient descent always find same solution?

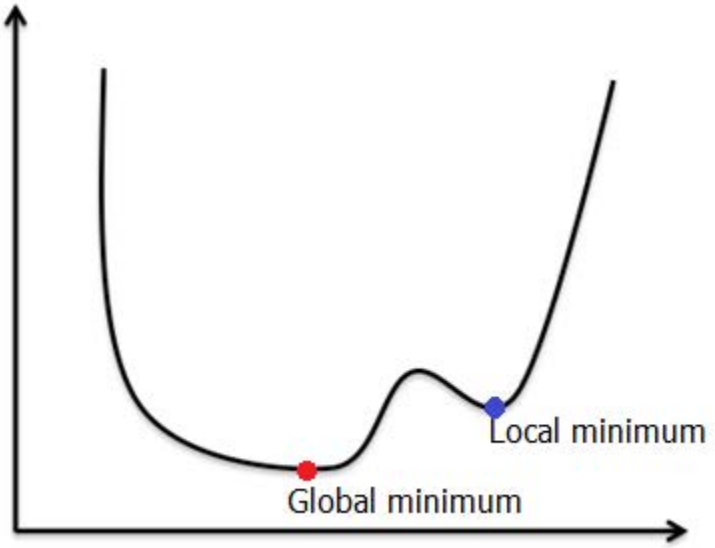




Will gradient descent always find same solution?

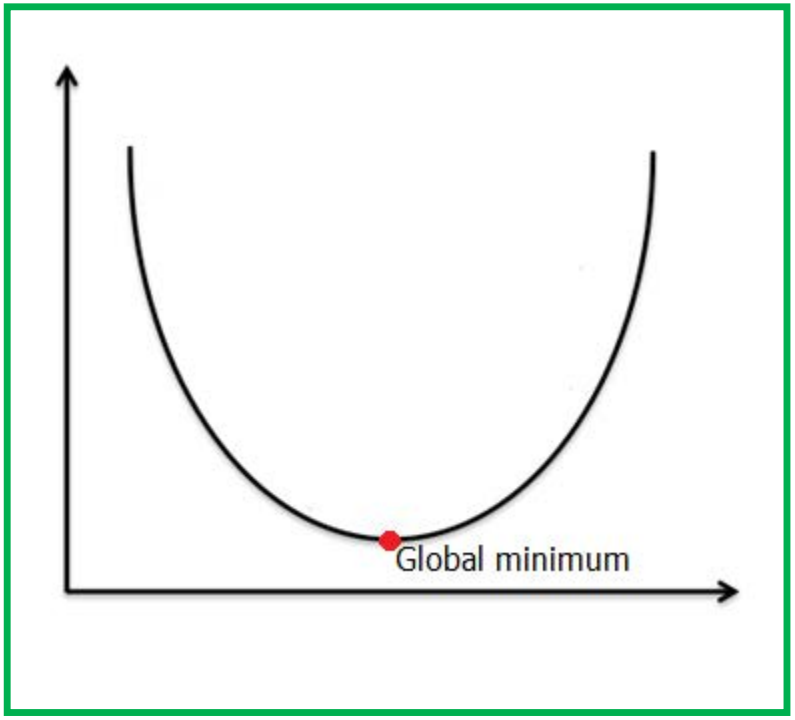


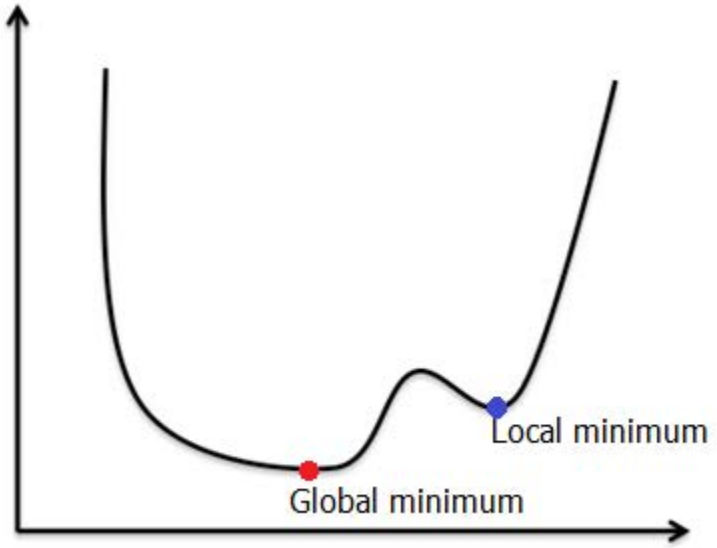
Yes, if loss looks like this



Not if multiple local minima exist

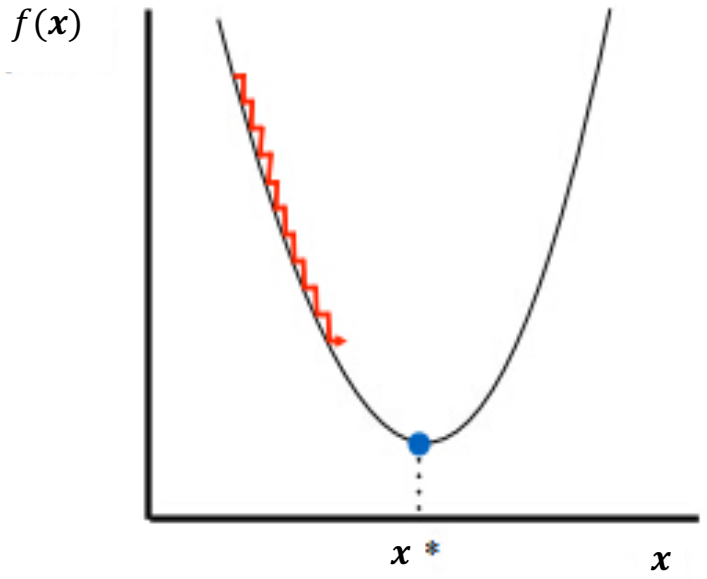
Log loss is convex!



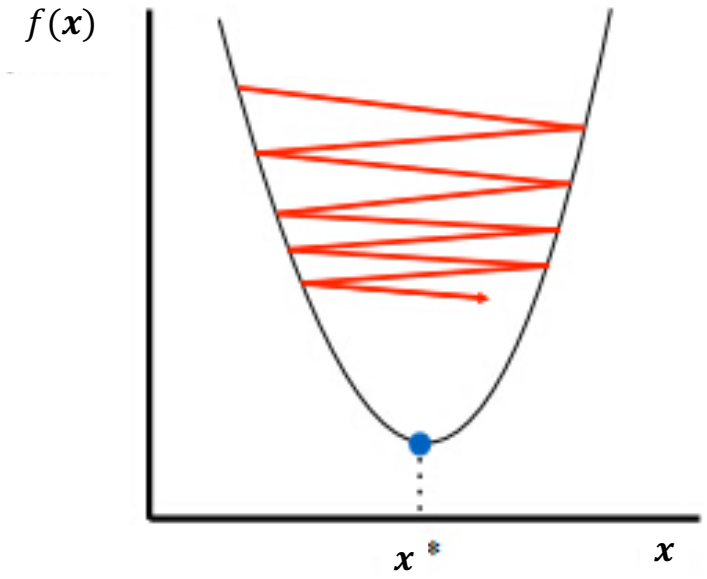


Intuition: 1D gradient descent

Choosing good step size matters!



Too small: converge very slowly



Too big: overshoot and even diverge

Log likelihood vs iterations

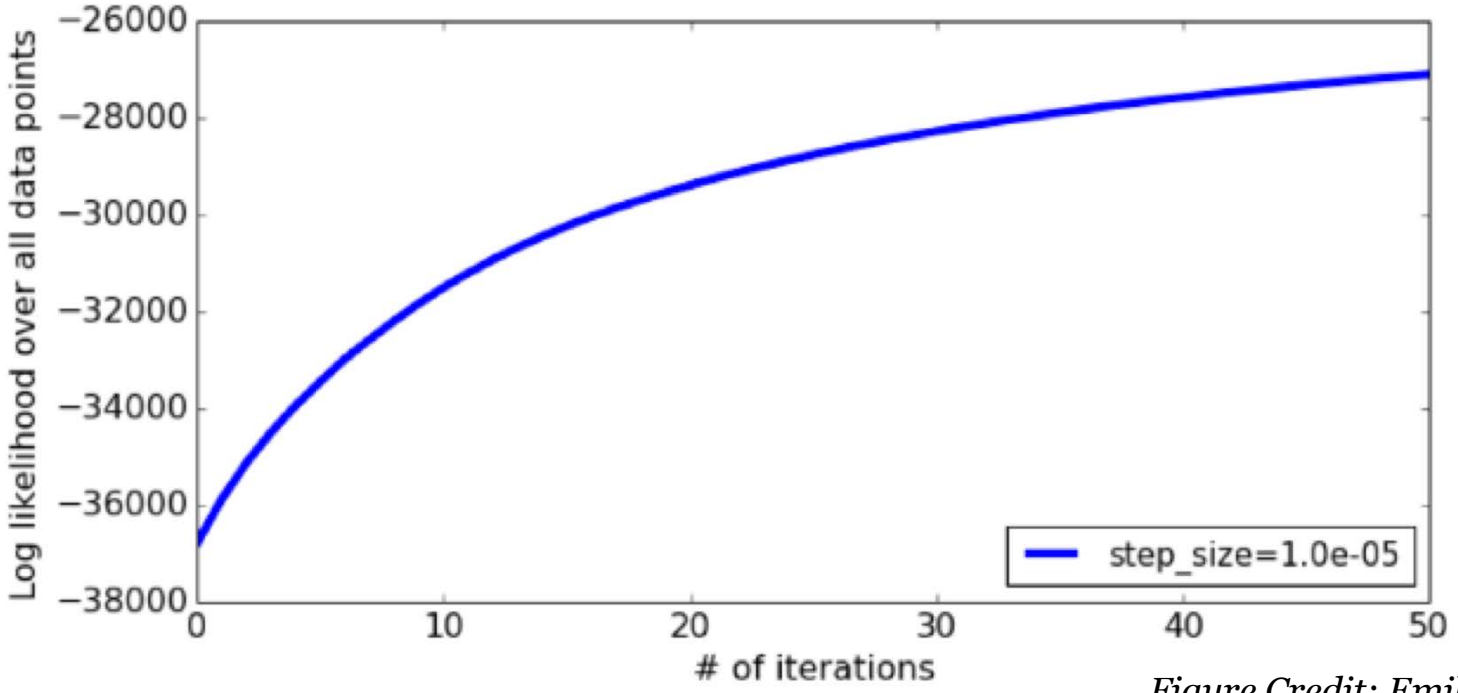


Figure Credit: Emily Fox (UW)

Maximizing likelihood: Higher is better!
(could multiply by -1 and minimize instead)

If step size is **too small**

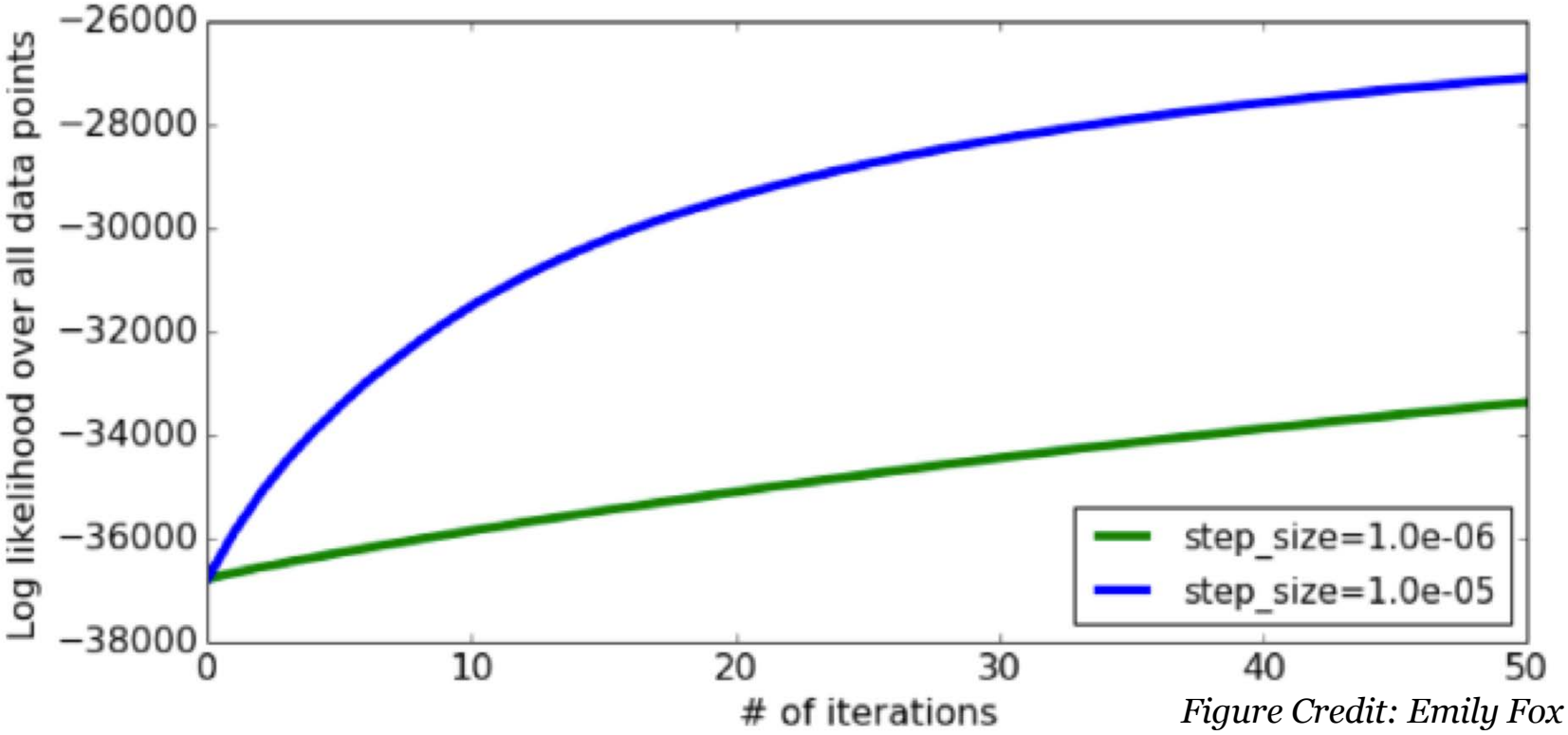


Figure Credit: Emily Fox (UW)

If step size is large

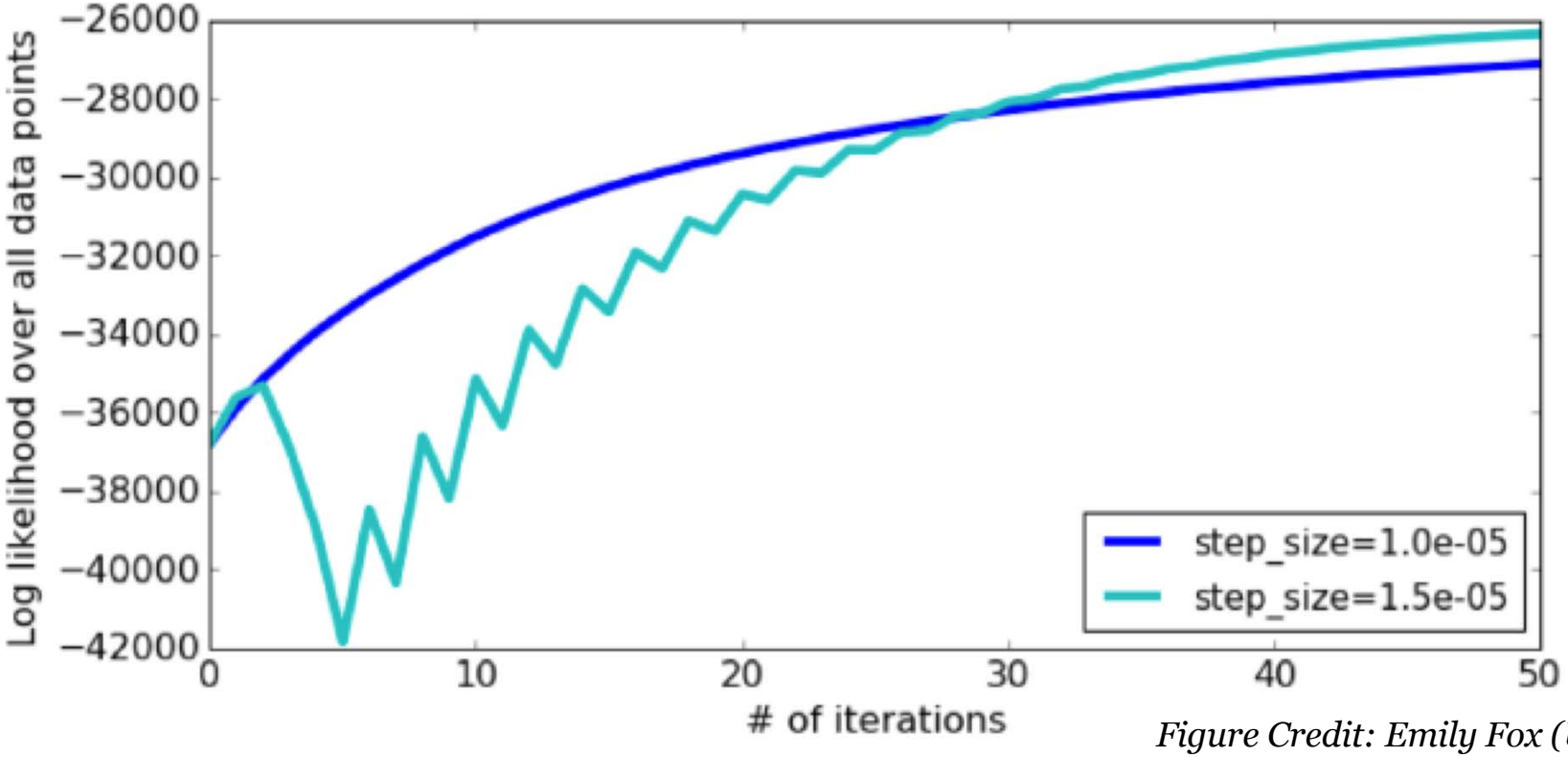


Figure Credit: Emily Fox (UW)

If step size is **too large**

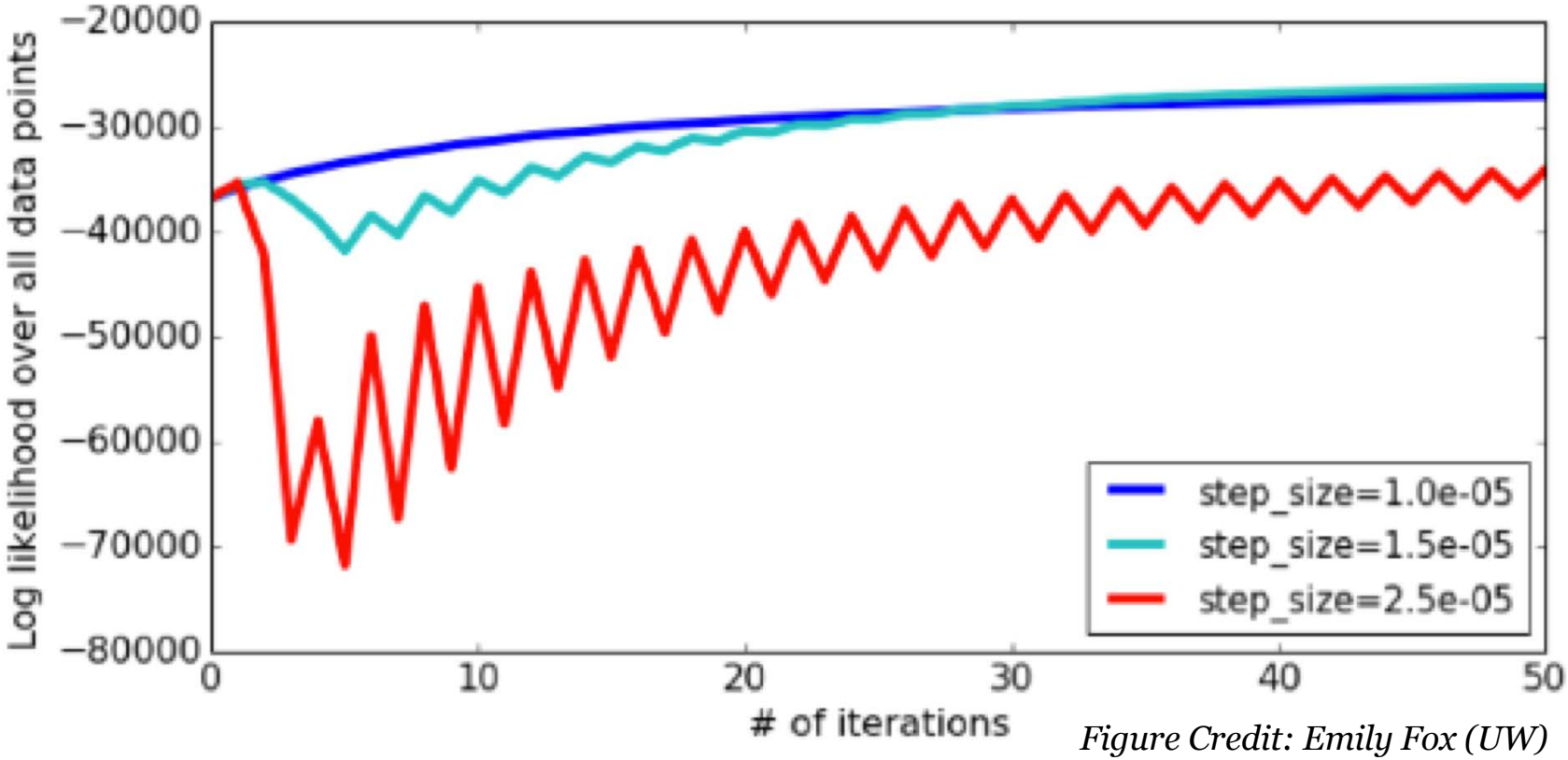


Figure Credit: Emily Fox (UW)

If step size is **way too large**

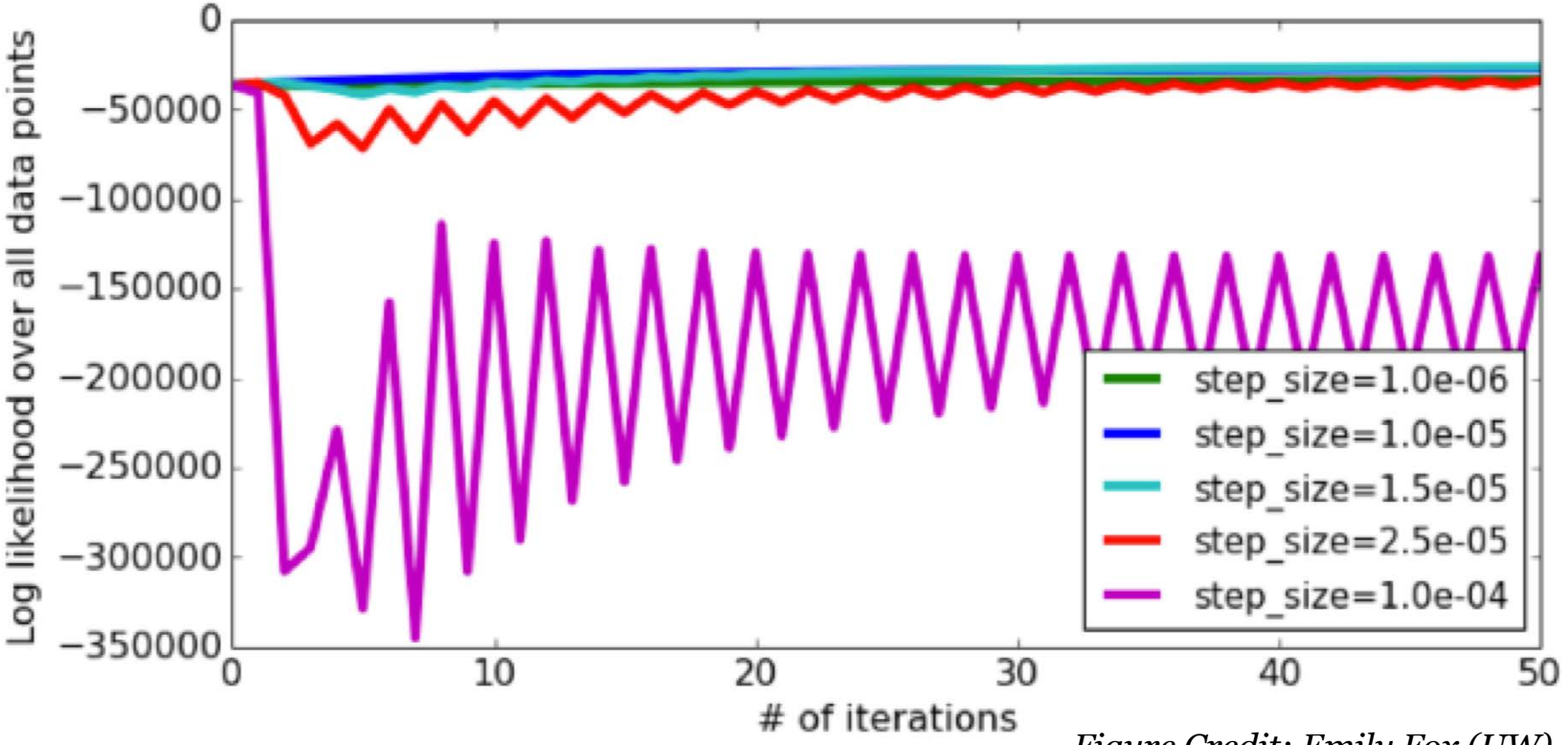


Figure Credit: Emily Fox (UW)

Rule for picking step sizes

- Never try just one!
- Try several values (exponentially spaced) until
  - Find one clearly too small
  - Find one clearly too large (oscillation / divergence)
- Always make trace plots!
  - Show the loss, norm of gradient, and parameters

- Smarter choices for step size:
  - Decaying methods
  - Search methods
  - Second-order methods

Decaying step sizes

input: initial $w \in \mathbb{R}$

input: initial step size $s_0 \in \mathbb{R}_+$

while not converged:

$$w \leftarrow w - s_t \nabla_w \mathcal{L}(w)$$

$$s_t \leftarrow \text{decay}(s_0, t)$$

$$t \leftarrow t + 1$$

Linear decay

$$\frac{s_0}{kt}$$

Exponential decay

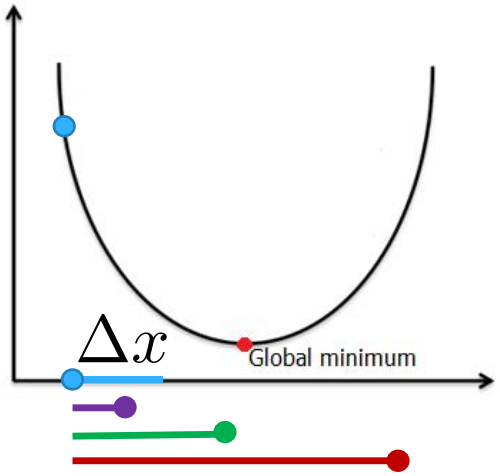
$$s_0 e^{-kt}$$

Often helpful, but hard to get right!

Searching for good step size

Goal: $\min_x f(x)$

Step Direction: $\Delta x = -\nabla_x f(x)$



Possible step lengths

Exact Line Search: Expensive but gold standard

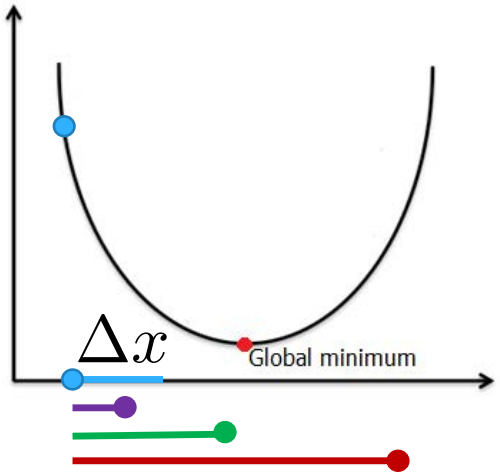
Search for the best scalar s >= 0, such that:

$$s^* = \arg\min_{s \geq 0} f(x + s\Delta x)$$

Searching for good step size

Goal: $\min_x f(x)$

Step Direction: $\Delta x = -\nabla_x f(x)$



Possible step lengths

Backtracking Line Search: More Efficient!

$s = 1$

while reduced slope linear extrapolation $\hat{f}(x + s\Delta x) < f(x + s\Delta x)$:

$s \leftarrow 0.9 \cdot s$

Backtracking line search

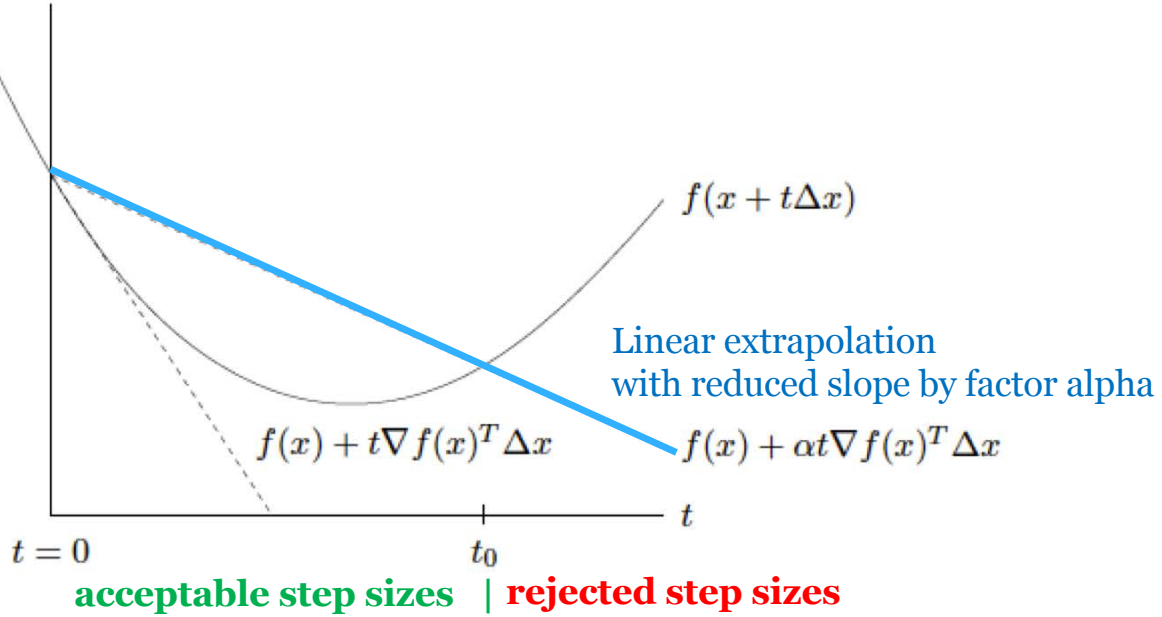


Figure 9.1 *Backtracking line search.* The curve shows f, restricted to the line over which we search. The lower dashed line shows the linear extrapolation of f, and the upper dashed line has a slope a factor of α smaller. The backtracking condition is that f lies below the upper dashed line, *i.e.*, $0 \leq t \leq t_0$.

$s = 1$

while reduced slope linear extrapolation $\hat{f}(x + s\Delta x) < f(x + s\Delta x)$:

$s \leftarrow 0.9 \cdot s$

Backtracking line search

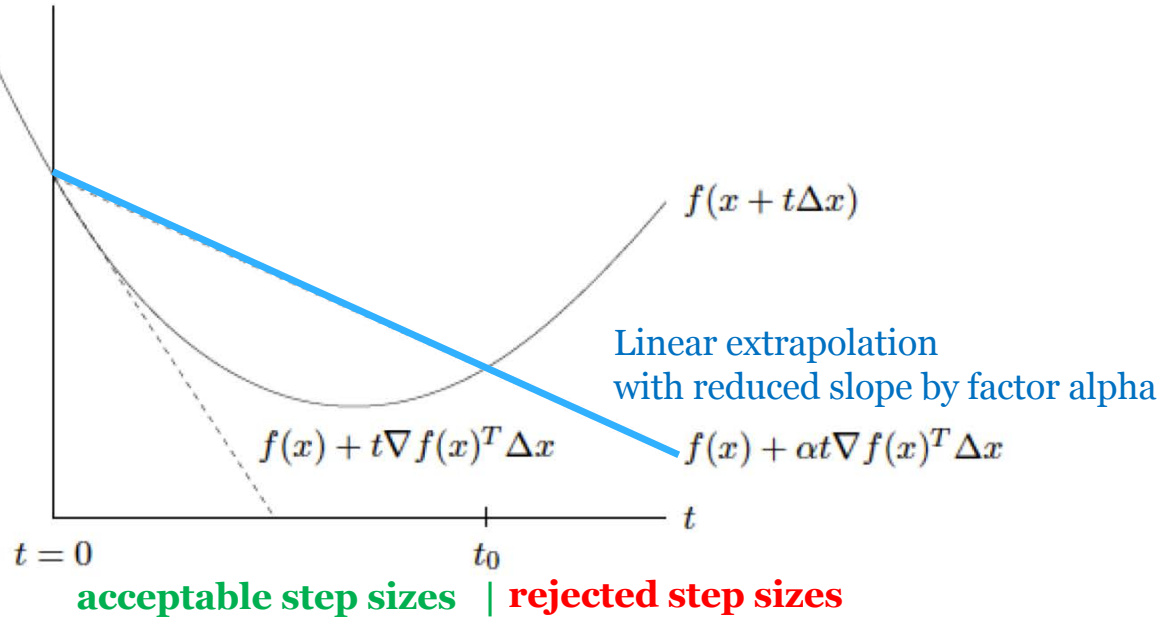


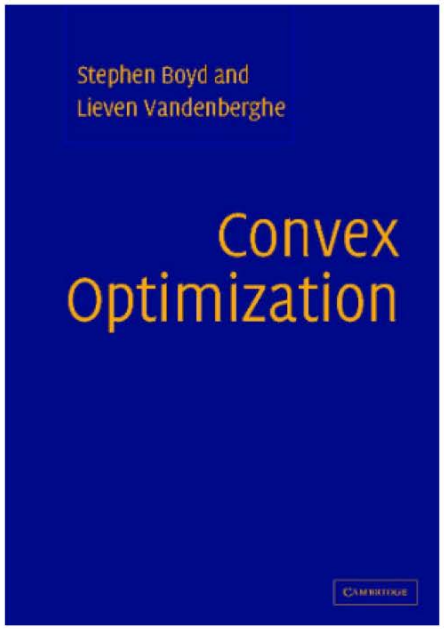
Figure 9.1 *Backtracking line search.* The curve shows f, restricted to the line over which we search. The lower dashed line shows the linear extrapolation of f, and the upper dashed line has a slope a factor of α smaller. The backtracking condition is that f lies below the upper dashed line, *i.e.*, $0 \leq t \leq t_0$.

In Python code: scipy.optimize.line_search

More resources on step sizes!

Online Textbook: Convex Optimization

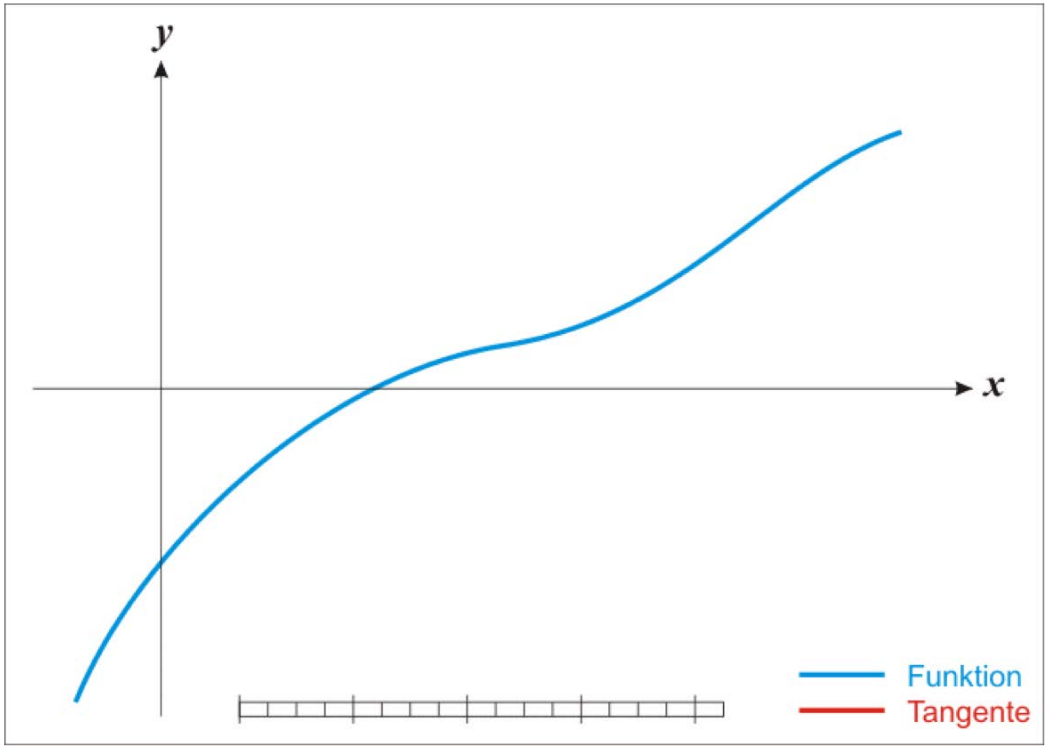
http://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf



Convex Optimization
Stephen Boyd and Lieven Vandenberghe

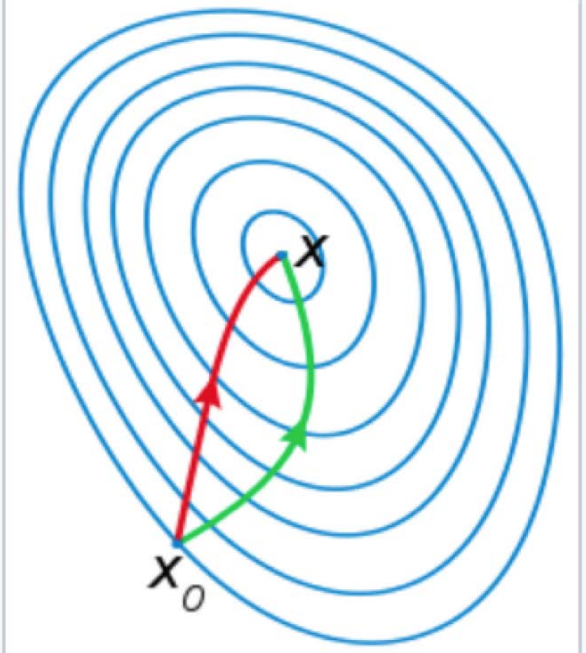
Cambridge University Press

Remember Newton's Method
Goal: find zero crossing



To optimize, we want to find zeros of first derivative!

Big Idea: 2nd deriv. can help!

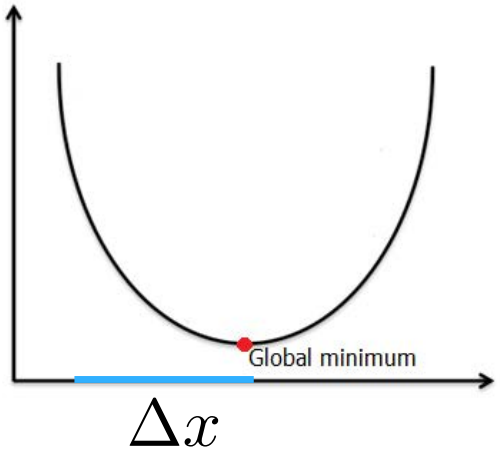


A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes). Newton's method uses curvature information (i.e. the second derivative) to take a more direct route.

Use second-derivative to rescale step size!

Goal: $\min_x f(x)$

Step Direction: $\Delta x = -\frac{f'(x_n)}{f''(x_n)}$



Will step directly to minimum if f is quadratic!

$$\Delta x = -H(x)^{-1} \nabla_x f(x)$$

In high dimensions, need the Hessian matrix

L-BFGS: the gold standard

scipy.optimize.fmin_l_bfgs_b

- Provide loss and gradient functions
- Approximates the Hessian via recent history of gradient steps

$$\Delta x = -H(x)^{-1} \nabla_x f(x)$$

In high dimensions, need the Hessian matrix
But this is quadratic in length of x , **expensive**

$$\Delta x = -\hat{H}(x)^{-1} \nabla_x f(x)$$

Instead, use low-rank approximation

L-BFGS : Limited Memory Broyden–Fletcher–Goldfarb–Shanno (BFGS)

Stochastic Gradient Descent

How can we go fast on big data?

Stochastic Estimate of Loss Function

- Standard "full-dataset" objective

$$\mathcal{L}(w) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n(x_n, y_n, w)$$

- Rewrite as "expected value" of empirical distrib."

$$\mathcal{L}(w) = \mathbb{E}_{x_i, y_i \sim \text{Unif}(\{x_n, y_n\}_{n=1}^{N})} \left[\mathcal{L}_i(x_i, y_i, w) \right]$$

Stochastic Estimate of Loss Function

- Standard "full-dataset" objective

$$\mathcal{L}(w) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n(x_n, y_n, w)$$

- Rewrite as "expected value" of empirical distrib."

$$\mathcal{L}(w) = \mathbb{E}_{x_i, y_i \sim \text{Unif}(\{x_n, y_n\}_{n=1}^N)} \left[\mathcal{L}_i(x_i, y_i, w) \right]$$

- Use one Monte Carlo sample to approximate:

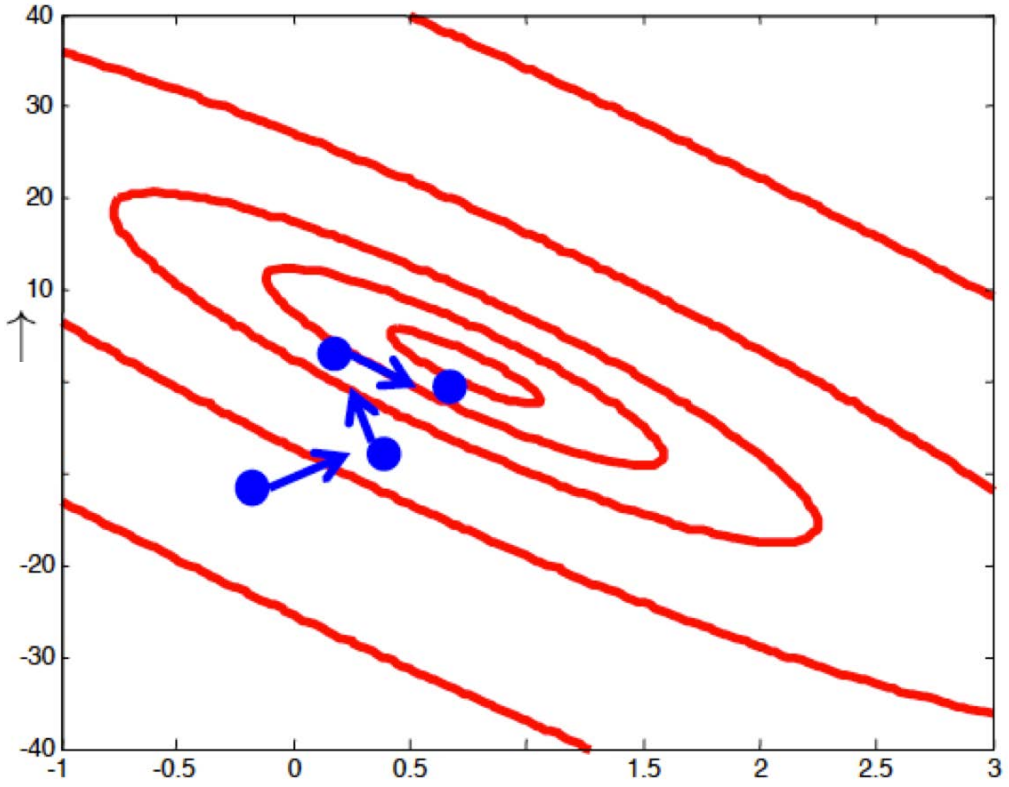
$$\mathcal{L}(w) \approx \mathcal{L}_i(x_i, y_i, w) \qquad x_i, y_i \sim \text{Unif}(\{x_n, y_n\}_{n=1}^N)$$

Stochastic Estimate of Gradient

Can use one Monte Carlo sample to approximate:

$$\nabla_w \mathcal{L}(w) \approx \nabla_w \mathcal{L}_i(x_i, y_i, w) \qquad x_i, y_i \sim \text{Unif}(\{x_n, y_n\}_{n=1}^N)$$

Intuition: Follow noisy but unbiased estimates of gradient



Stochastic gradient descent

input: initial $w \in \mathbb{R}$

input: step size $s \in \mathbb{R}_+$

while not converged:

$$\{x_i, y_i\} \sim \text{Unif}(\{x_n, y_n\}_{n=1}^N)$$

$$w \leftarrow w - s \nabla_w \mathcal{L}(x_i, y_i, w)$$

Shown here: one example at a time
Can also do: one minibatch at a time

Stochastic Gradient Descent

Benefits

- Faster (in terms of loss minimized per compute)
- More effective use of data
  - Don't wait to see all data before updating model

Limitations

- More parameters to tune (how many batches?)
- Descent is not guaranteed with every step
- Stopping conditions harder to evaluate
  - Can use running average of loss