CPSC 340: Machine Learning and Data Mining

Stochastic Gradient

Summer 2021

Admin

- Midterm grades are out
  - Submit regrade request on Gradescope
    - Not Piazza. Turnaround time will get longer if you do this
- Assignment 5 due Friday
- Assignment 6 out Friday
- Assignment 7 (optional) in the works
  - No due date, posted after final
  - Office hours by request
  - Will cover differentiable programming and deep learning

In This Lecture

1. Polynomial and Gaussian RBF Kernels (15 minutes)
2. Stochastic Gradient Descent (40 minutes)

Last Time: The “Other” Normal Equations

L2-reg. least squares

$$v = Z^T(ZZ^T + \lambda I)^{-1} y$$

$$\hat{y} = \tilde{Z} v$$

$$= \underbrace{\tilde{Z} Z^T}_{\tilde{K}} (\underbrace{ZZ^T}_{K} + \lambda I)^{-1} y$$

$$t \times 1 \quad = \underbrace{\tilde{K}}_{t \times n} (\underbrace{K + \lambda I}_{n \times n})^{-1} \underbrace{y}_{n \times 1}$$

- “kernel trick”: for certain bases (like polynomials), We can efficiently compute K and $\tilde{K}$ even though forming Z and $\tilde{Z}$ is intractable.
  - In the same way we can comptue $(x+1)^9$ instead of $x^9 + 9x^8 + 36x^7 + 84x^6$…

Last Time: Degree-2 Kernel

- Consider two examples x_i and x_j for a 2-dimensional dataset:

$$x_i = (x_{i1}, x_{i2}) \qquad x_j = (x_{j1}, x_{j2})$$

- Now consider a particular degree-2 basis:

$$z_i = (x_{i1}^2, \sqrt{2}\, x_{i1} x_{i2}, x_{i2}^2) \qquad z_j = (x_{j1}^2, \sqrt{2}\, x_{j1} x_{j2}, x_{j2}^2)$$

- In this case the inner product $z_i^T z_j$ is $k(x_i,x_j) = (x_i^T x_j)^2$:

[1] $$z_i^T z_j = x_{i1}^2 x_{j1}^2 + (\sqrt{2}\, x_{i1} x_{i2})(\sqrt{2}\, x_{j1} x_{j2}) + x_{i2}^2 x_{j2}^2$$

[2] $$= x_{i1}^2 x_{j1}^2 + 2 x_{i1} x_{i2} x_{j1} x_{j2} + x_{i1}^2 x_{i2}^2$$

[3] $$= (\underbrace{x_{i1} x_{j1} + x_{i2} x_{j2}}_{x_i^T x_j})^2$$ "completing the square"

[4] $$= (x_i^T x_j)^2$$ ← No <u>need</u> for z_i to compute $z_i^T z_j$

Coming Up Next

POLYNOMIAL AND GAUSSIAN RBF KERNELS

Polynomial Kernel with Higher Degrees

- Let’s add a bias and linear terms to our degree-2 basis:

$$z_i = \begin{bmatrix} 1 & \sqrt{2}x_{i1} & \sqrt{2}x_{i2} & x_{i1}^2 & \sqrt{2}x_{i1}x_{i2} & x_{i2}^2 \end{bmatrix}^T$$

- In this case the inner product $z_i^Tz_j$ is $k(x_i,x_j) = (1 + x_i^Tx_j)^2$:

[1] $$(1 + x_i^Tx_j)^2 = 1 + 2x_i^Tx_j + (x_i^Tx_j)^2$$

[2] $$= 1 + 2x_{i1}x_{j1} + 2x_{i2}x_{j2} + x_{i1}^2x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} + x_{i2}^2x_{j2}^2$$

[3] $$= \underbrace{\begin{bmatrix} 1 & \sqrt{2}x_{i1} & \sqrt{2}x_{i2} & x_{i1}^2 & \sqrt{2}x_{i1}x_{i2} & x_{i2}^2 \end{bmatrix}}_{z_i^T} \underbrace{\begin{bmatrix} 1 \\ \sqrt{2}x_{j1} \\ \sqrt{2}x_{j2} \\ x_{j2}^2 \\ \sqrt{2}x_{j1}x_{j2} \\ x_{j2}^2 \end{bmatrix}}_{z_j}$$

[4] $$= z_i^Tz_j$$

Polynomial Kernel with Higher Degrees

- To get all degree-4 "monomials" I can use:

$$k(x_i, x_j) = (x_i^T x_j)^4$$

Equivalent to using a z_i with weighted versions of $x_{i1}^4, x_{i1}^3 x_{i2}, x_{i1}^2 x_{i2}^2, x_{i1} x_{i2}^3, x_{i2}^4, \dots$

- To also get lower-order terms use $k(x_i,x_j) = (1 + x_i^T x_j)^4$
- The general degree-p polynomial kernel function:

$$k(\underset{d\times 1}{x_i}, \underset{d\times 1}{x_j}) = (1 + \underbrace{\underbrace{x_i^T x_j}_{1\times d \;\; d\times 1}}_{O(d)})^{p}$$

  - Works for any number of features 'd'.
  - But cost of computing one $k(x_i,x_j)$ is $O(d)$ instead of $O(d^p)$ to compute $z_i^T z_j$.
  - Take-home message: I can compute dot-products without the features.

Kernel Trick with Polynomials

- Using polynomial basis of degree ‘p’ with the kernel trick:
  - Compute K and $\tilde{K}$ using:

$$K_{ij} = (1 + x_i^T x_j)^p \qquad \tilde{K}_{ij} = (1 + \underbrace{\tilde{x}_i}_{\text{test example}}{}^T \underbrace{x_j}_{\text{train example}})^p$$

  - Make predictions using:

$k(x_i, x_j) \rightarrow O(d)$

$$\begin{array}{cc} & n \\ n & \begin{bmatrix} O & \\ & \end{bmatrix} \end{array}$$

$$\underbrace{\hat{y}}_{t \times 1} = \underbrace{\tilde{K}}_{t \times n} \underbrace{(K + \lambda I)^{-1}}_{n \times n} \underbrace{y}_{n \times 1} = \tilde{K} u$$

$u = (K + \lambda I)^{-1} y$

- Training cost is only O($n^2 d + n^3$), despite using k=O(d^p) features.
  - We can form ‘K’ in O(n²d), and we need to “invert” an ‘n x n’ matrix.
  - Testing cost is only O(tnd), cost to form $\tilde{K}$.

Gaussian-RBF Kernel

- Most common kernel is the Gaussian RBF kernel:

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

$$\underset{n\times n}{K} \qquad \underset{t\times n}{\tilde{K}}$$

- Same formula and behaviour as RBF basis, but not equivalent:
  - Before we used RBFs as a basis, now we're using them as inner-product.

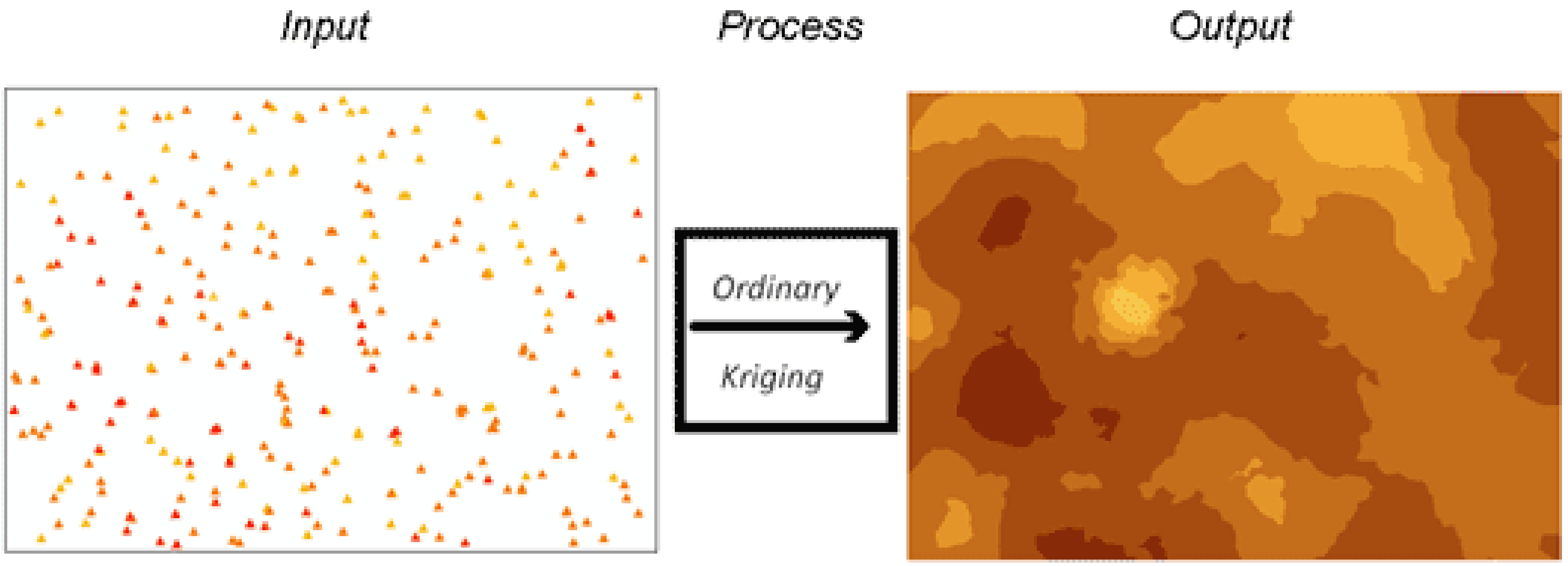
$$\underset{d\times 1}{x_i} \rightarrow \underset{\infty\times 1}{z_i} \qquad \underset{d\times 1}{x_j} \rightarrow \underset{\infty\times 1}{z_j} \qquad \underset{1\times\infty \quad \infty\times 1}{z_i^T z_j}$$

- Basis z_i giving Gaussian RBF kernel is infinite-dimensional.
  - If d=1 and σ=1, it corresponds to using this basis (bonus slide):

$$z_i = \exp(-x_i^2)\left[1 \quad \sqrt{\frac{2}{1!}}x_i \quad \sqrt{\frac{2^2}{2!}}x_i^2 \quad \sqrt{\frac{2^3}{3!}}x_i^3 \quad \sqrt{\frac{2^4}{4!}}x_i^4 \quad \cdots\cdots\right]$$

Motivation: Finding Gold

- Kernel methods first came from mining engineering (“Kriging”):
  - Mining company wants to find gold.
  - Drill holes, measure gold content.
  - Build a kernel regression model (typically use RBF kernels).



http://www.bisolutions.us/A-Brief-Introduction-to-Spatial-Interpolation.php

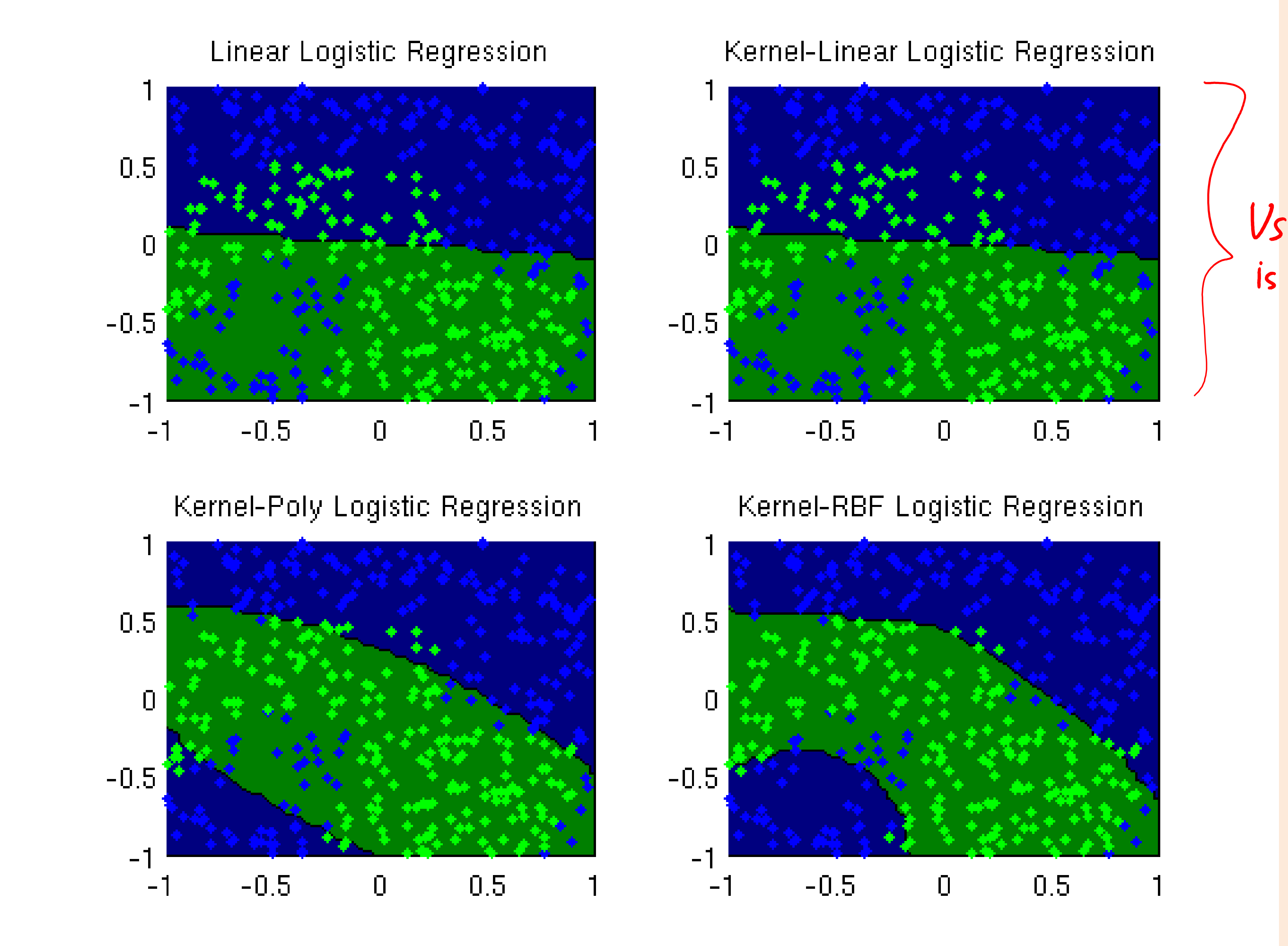
Kernel Trick for Other Methods

- Besides L2-regularized least squares, when can we use kernels?
  - We can compute Euclidean distance with kernels:

$$\|z_i - z_j\|^2 = z_i^T z_i - 2 z_i^T z_j + z_j^T z_j = k(x_i, x_i) - 2k(x_i, x_j) + k(x_j, x_j)$$

  - All of our distance-based methods have kernel versions:
    - Kernel k-nearest neighbours.
    - Kernel k-means clustering (allows non-convex clusters)
    - Kernel density-based clustering.
    - Kernel hierarchical clustering.
    - Kernel distance-based outlier detection.
    - Kernel “Amazon Product Recommendation”.

Logistic Regression with Kernels



Using "linear" kernel is the same as using original features

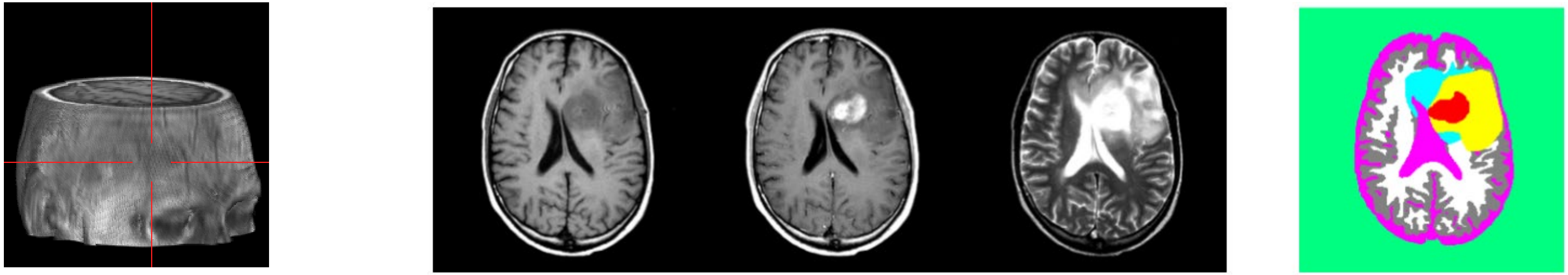
When you use SGD
but don’t tune step size

Coming Up Next

STOCHASTIC GRADIENT DESCENT INTRO

Motivation: Big-n Problems

- Recall the automatic brain tumour segmentation problem:



- MRI scanners at the time produced 200x200x200 volumes.
  - So one scan gives 8 million examples.
  - And you need to train on more than one scan!

- Similar issues arise in the Gmail application:
  - If every email is a training example, you have LOTS of training examples.

Motivation: Big-n Problems

- Consider fitting a least squares model:

$$f(w) = \frac{1}{2}\sum_{i=1}^{n}(w^Tx_i - y_i)^2 \propto \frac{1}{n}\sum_{i=1}^{n}(w^Tx_i - y_i)^2$$

- Gradient methods are effective when 'd' is very large.
  - O(nd) per iteration instead of O($nd^2 + d^3$) to solve as linear system.

- But what if number of training examples 'n' is very large?
  - All Gmails, all products on Amazon, all homepages, all images, etc.

Gradient Descent vs. Stochastic Gradient

- Recall the gradient descent algorithm:

$$w^{t+1} = w^t - \alpha^t \nabla f(w^t)$$

- For least squares, our gradient has the form:

$$\underbrace{\nabla f(w)}_{d \times 1} = \sum_{i=1}^{n} \underbrace{(w^T x_i - y_i)}_{\text{scalar}} \underbrace{x_i}_{d \times 1}$$

- So the cost of computing this gradient is linear in ‘n’.
    - As ‘n’ gets large, gradient descent iterations become expensive.

Gradient Descent vs. Stochastic Gradient

- Common solution to this problem is stochastic gradient algorithm:

$$w^{t+1} = w^t - \alpha^t \nabla f_i(w^t)$$

- Uses the gradient of a randomly-chosen training example:

$$\underbrace{\nabla f_i(w)}_{d\times 1} = (\underbrace{w^T x_i - y_i}_{\text{scalar}})\underbrace{x_i}_{d\times 1}$$

- Cost of computing this one gradient is $O(d)$.
  - Independent of ‘n’!
  - Iterations are ‘n’ times faster than gradient descent iterations.
  - With 1 billion training examples, this iteration is 1 billion times faster.

Stochastic Gradient (SG)

- Stochastic gradient is an iterative optimization algorithm:
  - We start with some initial guess, w^0.
  - Generate new guess by moving in the negative gradient direction:

$$w^1 = w^0 - \alpha^0 \nabla f_i(w^0)$$

    - For a random training example 'i'.
  - Repeat to refine the guess:

$$w^{t+1} = w^t - \alpha^t \nabla f_i(w^t) \quad \text{for } t = 1, 2, 3, \dots$$

    - For a random training example 'i'.

“Epoch”

- “Epoch” := number of stochastic gradient steps that amounts to using ‘n’ examples
  - Right now, one epoch is ‘n’ stochastic gradient steps
  - With mini-batches (later) of size B, one epoch is ‘n / B’ steps

- Important: ‘t’ denotes stochastic gradient step iteration
  - Not epoch iteration
  - t = n after first epoch, t = 2n after second epoch, etc.

① Pick completely random

② shuffle and for-loop

for i in range(n)

$\nabla f_{\underset{\uparrow}{i}}(w^t)$

Problem where we can use Stochastic Gradient

- Stochastic gradient applies when minimizing averages:

$$f(w) = \frac{1}{n}\sum_{i=1}^{n}(w^Tx_i - y_i)^2 \quad \text{(squared error)}$$

$$f(w) = \frac{1}{n}\sum_{i=1}^{n}\log(1+\exp(-y_iw^Tx_i)) \quad \text{(logistic regression)}$$

$$f(w) = \frac{1}{n}\sum_{i=1}^{n}\left[\log(1+\exp(-y_iw^Tx_i)) + \frac{\lambda}{2}\|w\|^2\right] \quad \text{(L_2-regularized logistic)}$$

$$f(w) = \frac{1}{n}\sum_{i=1}^{n}\underbrace{f_i}_{\text{error for example } i}(w) \quad \text{(our \underline{notation} for the general case)}$$

$$\nabla f(w) = \frac{1}{n}\sum_{i=1}^{n}\nabla f_i(w)$$

$$\|Xw-y\|_\infty = \max\{w^Tx_i - y_i\}$$

Q: What about brittle regression?

Why Does Stochastic Gradient Work / Not Work?

- Main problem with stochastic gradient:
  - Gradient of random example might point in the wrong direction.

- Does this have any hope of working?
  - The expected direction is the full gradient.

$$\underset{\substack{\uparrow \\ \text{expectation over} \\ \text{choice of random} \\ \text{example 'i'}}}{E[\nabla f_i(w^k)]} = \underbrace{\sum_{i=1}^{n} p(i)\nabla f_i(w^k)}_{\substack{\text{definition of} \\ \text{expectation}}} = \sum_{i=1}^{n} \underset{\substack{\uparrow \\ \text{if each example} \\ \text{is equally likely}}}{\frac{1}{n}} \nabla f_i(w^k) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w^k) = \underbrace{\nabla f(w^k)}_{\substack{\text{gradient over} \\ \underline{\text{all}} \text{ examples}}}$$

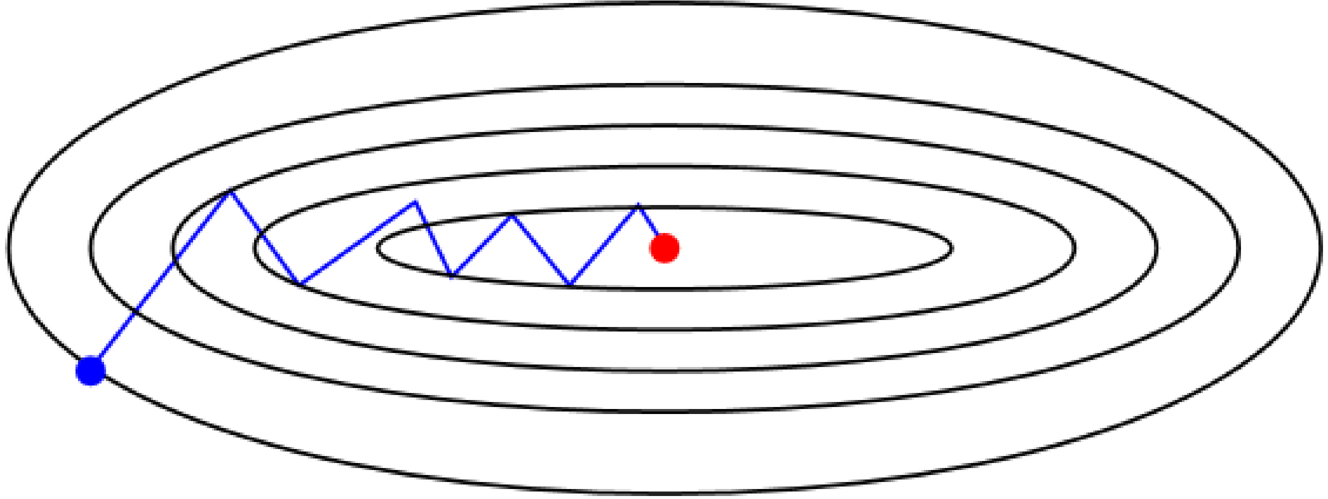
  - The algorithm is going in the right direction on average.

Coming Up Next

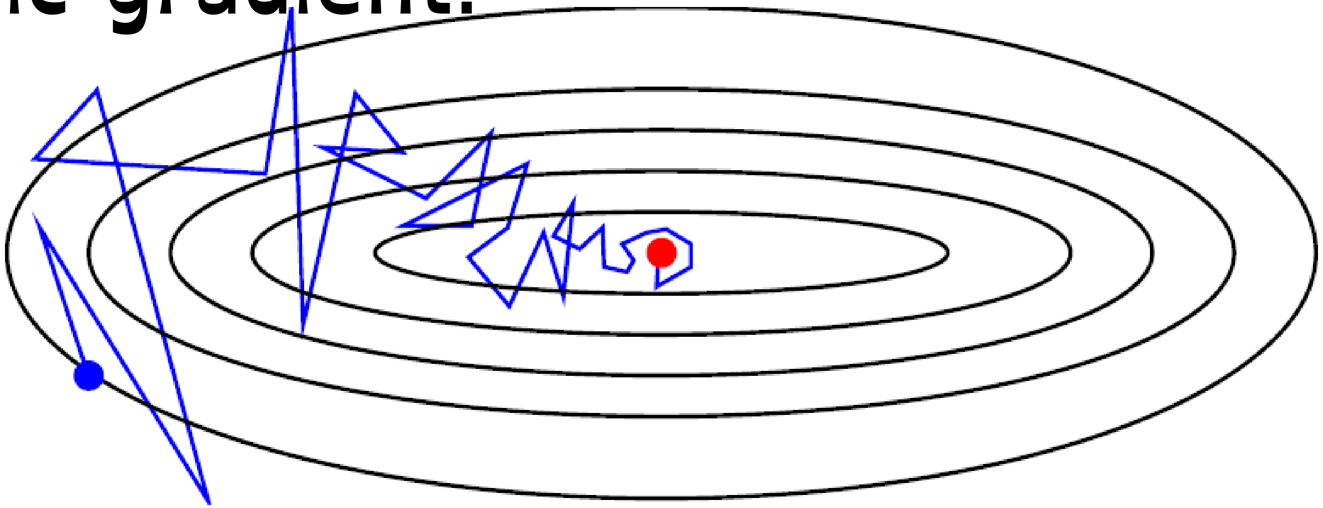
VISUAL EXPLANATION OF STOCHASTIC GRADIENT

Gradient Descent vs. Stochastic Gradient (SG)

- Gradient descent:

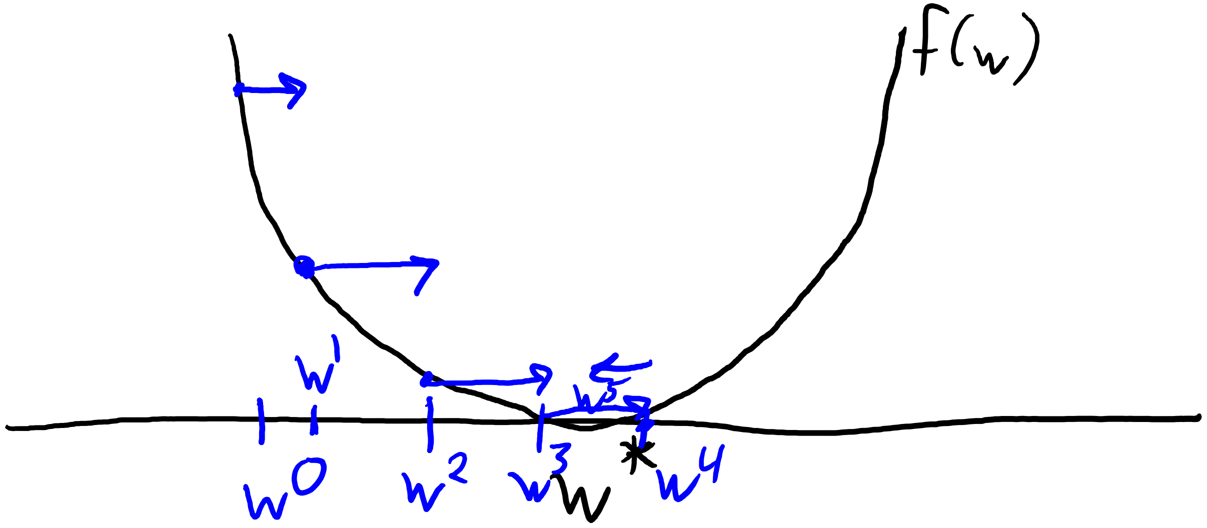


- Stochastic gradient:



Gradient Descent in Action

$$f(w) = \frac{1}{5}\sum_{i=1}^{5}(w^T x_i - y_i)^2$$



Stochastic Gradient in Action

$$f(w) = \frac{1}{5}\sum_{i=1}^{5}(w^T x_i - y_i)^2$$

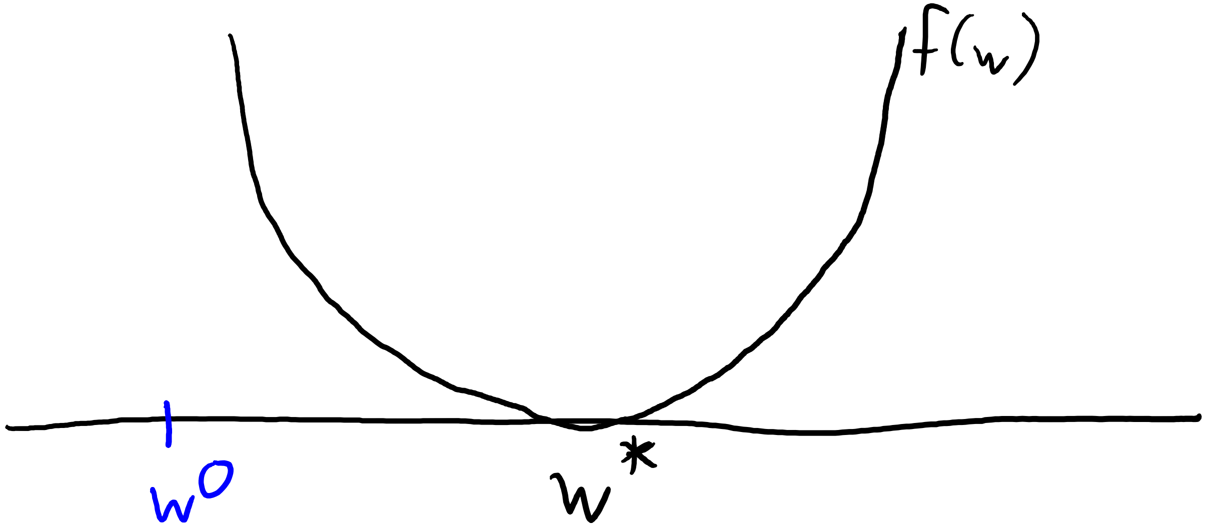
$$f_1(w) = (w^T x_1 - y_1)^2$$

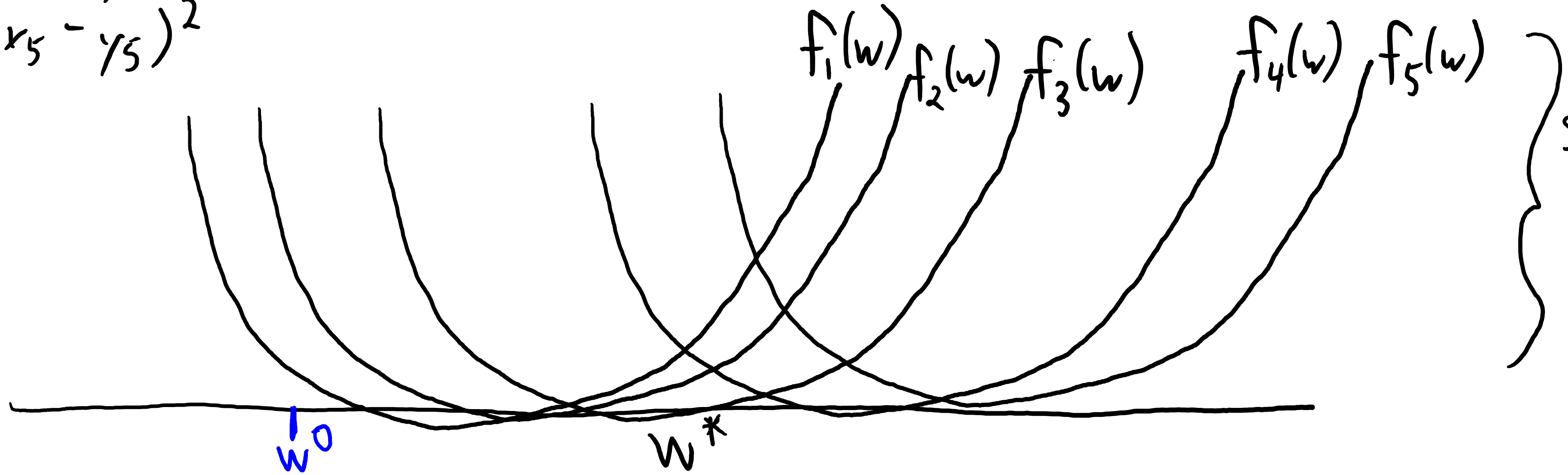
$$f_2(w) = (w^T x_2 - y_2)^2$$

$$f_3(w) = (w^T x_3 - y_3)^2$$

$$f_4(w) = (w^T x_4 - y_4)^2$$

$$f_5(w) = (w^T x_5 - y_5)^2$$





Stochastic gradient minimizes <u>average</u> value.

Stochastic Gradient in Action

$f(w) = \frac{1}{5}\sum_{i=1}^{5}(w^Tx_i - y_i)^2$

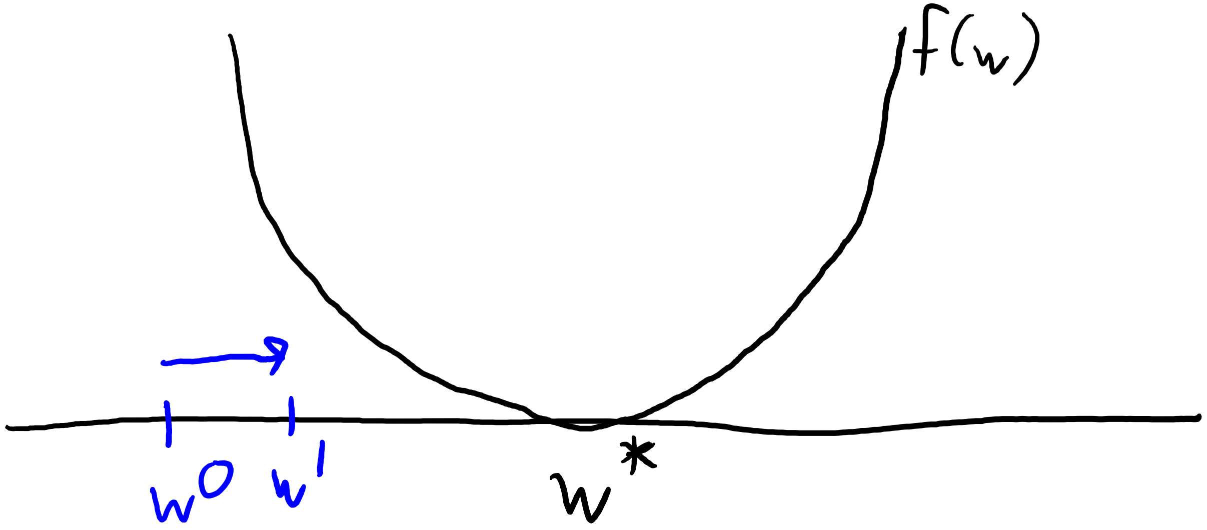
$f_1(w) = (w^Tx_1 - y_1)^2$

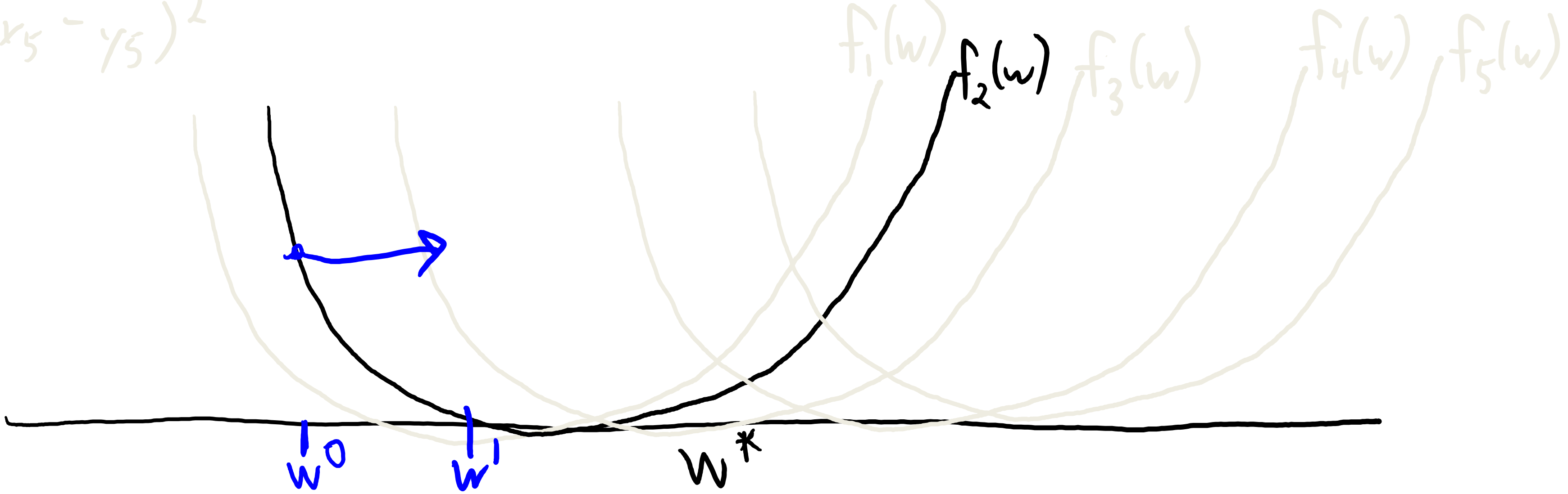
$f_2(w) = (w^Tx_2 - y_2)^2$

$f_3(w) = (w^Tx_3 - y_3)^2$

$f_4(w) = (w^Tx_4 - y_4)^2$

$f_5(w) = (w^Tx_5 - y_5)^2$





Stochastic gradient minimizes average value.

Stochastic Gradient in Action

$$f(w) = \frac{1}{5}\sum_{i=1}^{5} (w^T x_i - y_i)^2$$

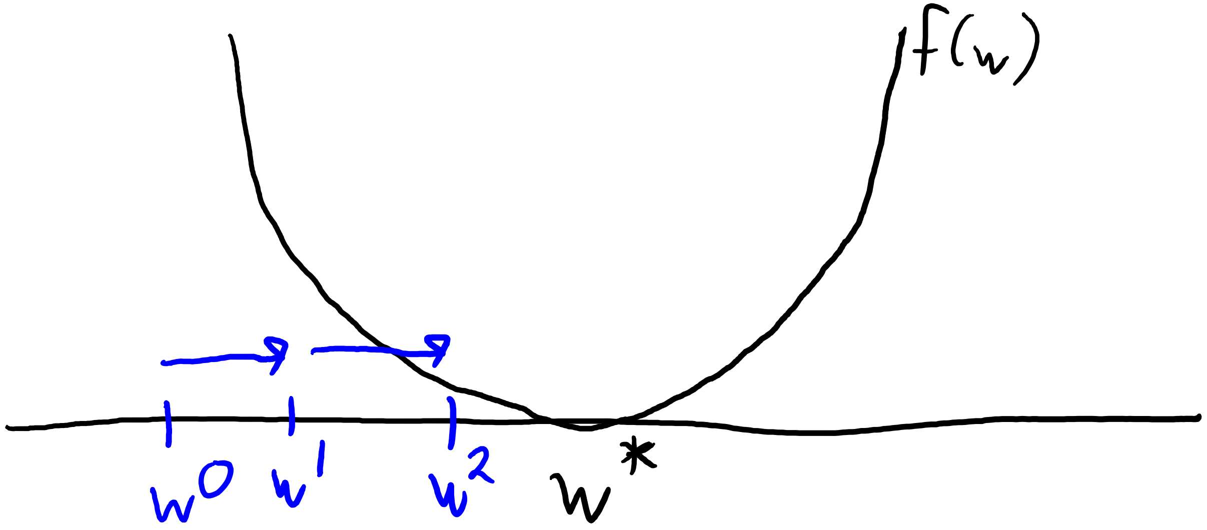
$$f_1(w) = (w^T x_1 - y_1)^2$$

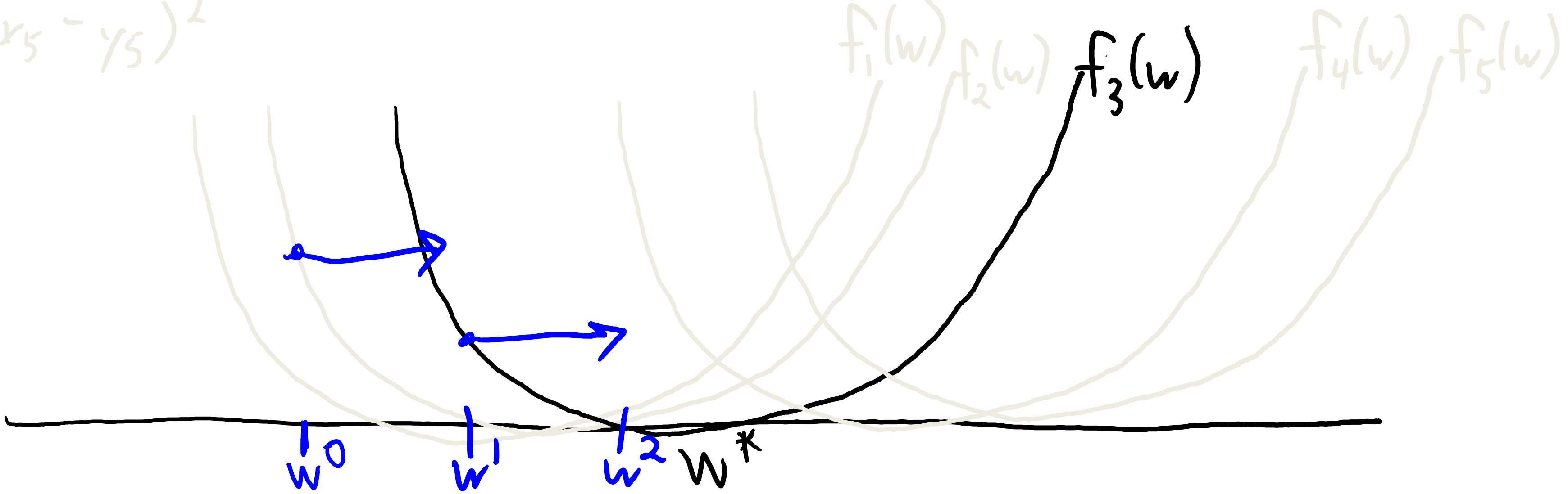
$$f_2(w) = (w^T x_2 - y_2)^2$$

$$f_3(w) = (w^T x_3 - y_3)^2$$

$$f_4(w) = (w^T x_4 - y_4)^2$$

$$f_5(w) = (w^T x_5 - y_5)^2$$





Stochastic gradient minimizes average value.

Stochastic Gradient in Action

$$f(w) = \frac{1}{5}\sum_{i=1}^{5}(w^T x_i - y_i)^2$$

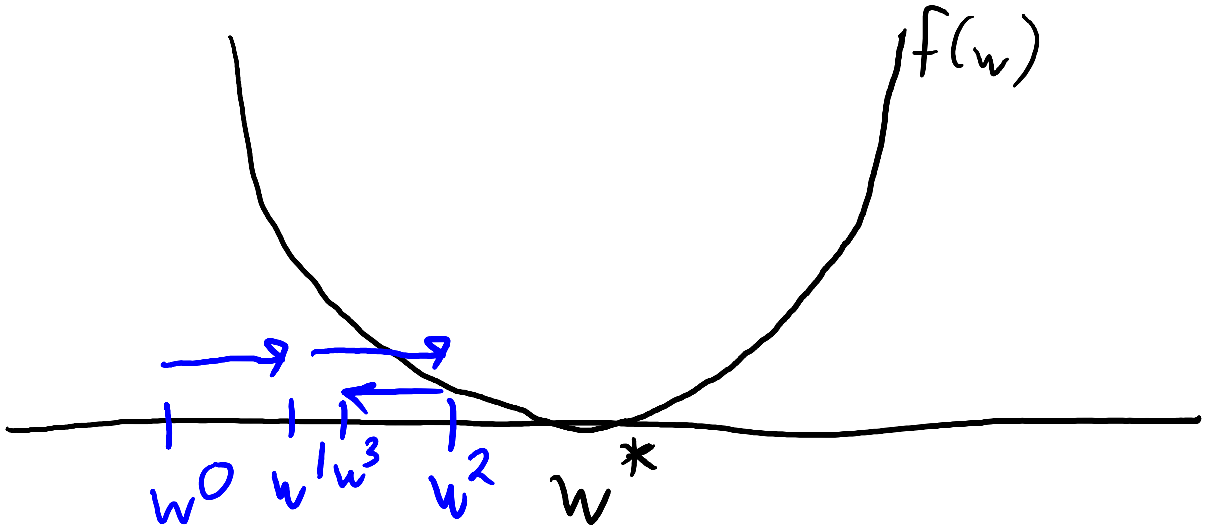
$$f_1(w) = (w^T x_1 - y_1)^2$$

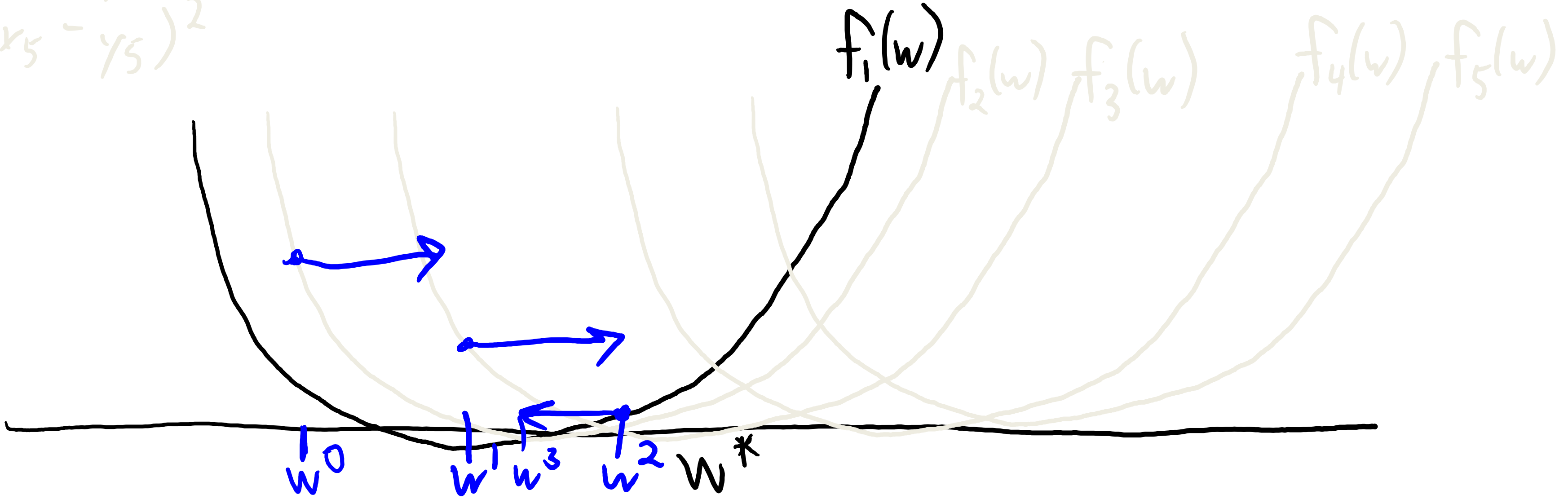
$$f_2(w) = (w^T x_2 - y_2)^2$$

$$f_3(w) = (w^T x_3 - y_3)^2$$

$$f_4(w) = (w^T x_4 - y_4)^2$$

$$f_5(w) = (w^T x_5 - y_5)^2$$





Stochastic gradient minimizes average value.

Stochastic Gradient in Action

$$f(w) = \frac{1}{5}\sum_{i=1}^{5}(w^T x_i - y_i)^2$$

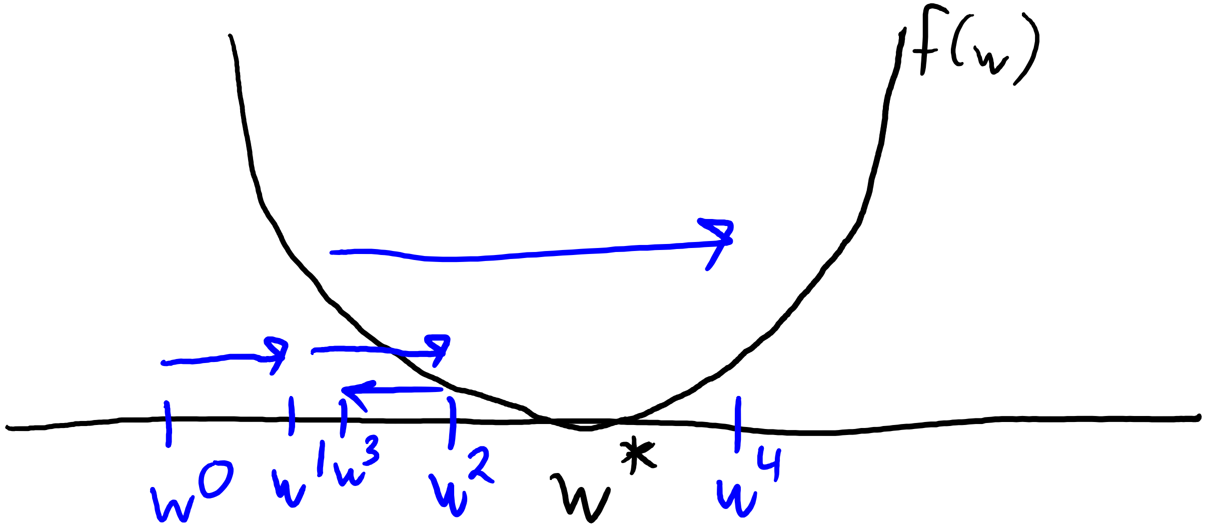
$$f_1(w) = (w^T x_1 - y_1)^2$$

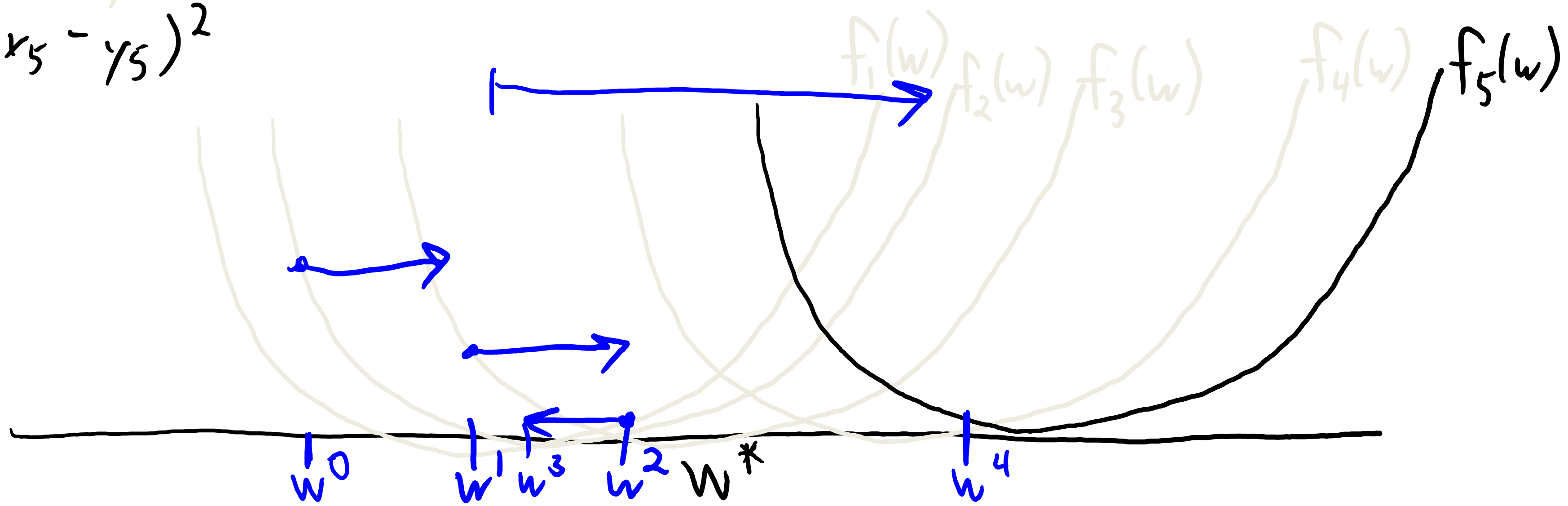
$$f_2(w) = (w^T x_2 - y_2)^2$$

$$f_3(w) = (w^T x_3 - y_3)^2$$

$$f_4(w) = (w^T x_4 - y_4)^2$$

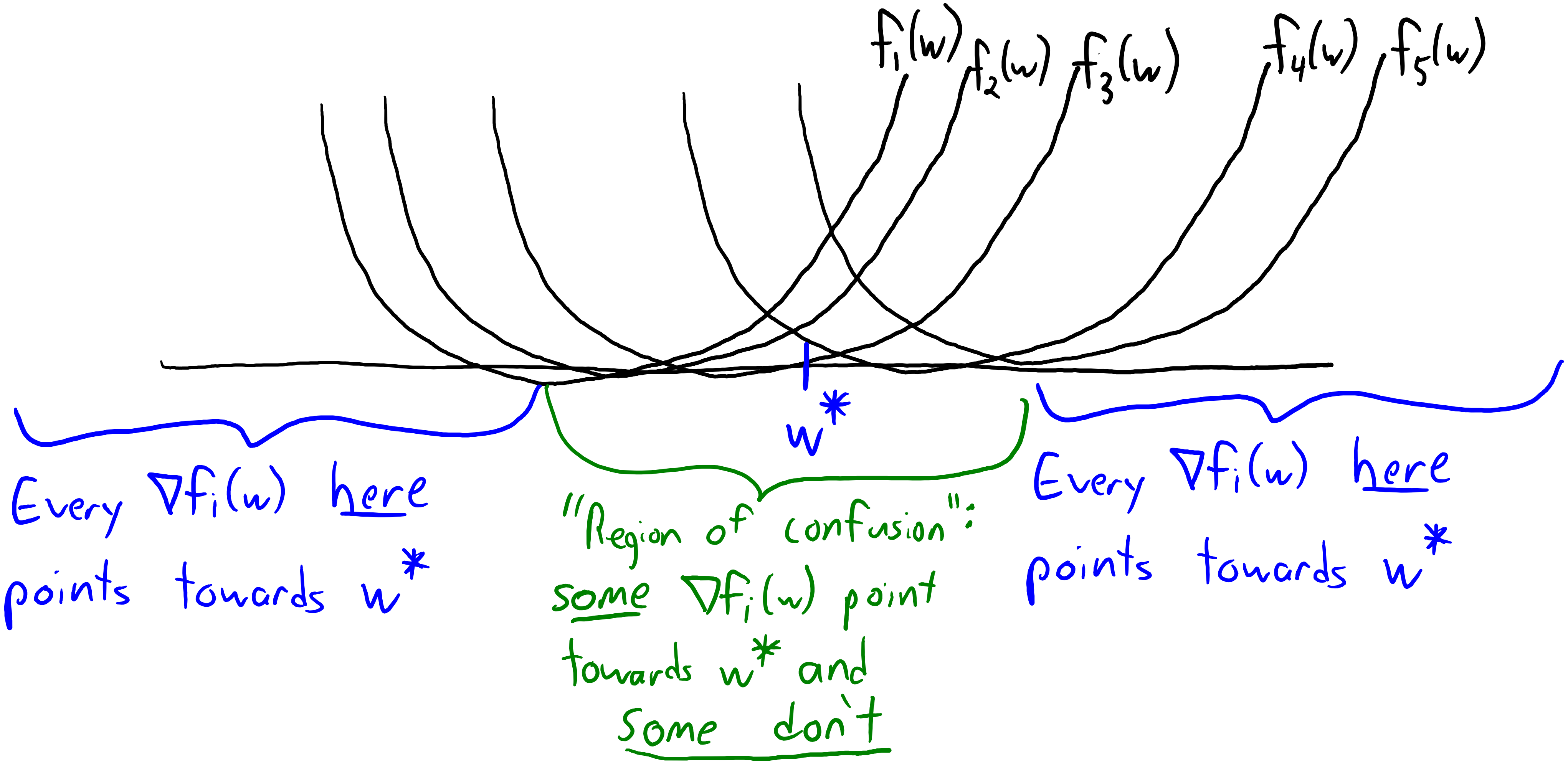
$$f_5(w) = (w^T x_5 - y_5)^2$$





Stochastic gradient minimizes average value.

Effect of ‘w’ Location on Progress



Every $\nabla f_i(w)$ here points towards w^*

"Region of confusion": some $\nabla f_i(w)$ point towards w^* and some don't

Every $\nabla f_i(w)$ here points towards w^*

- We’ll still make good progress if “most” gradients points in right direction.

When you set step size to 3e-4 and SGD just works

Coming Up Next

STEP SIZES OF STOCHASTIC GRADIENT

Variance of the Random Gradients

- The “confusion” is captured by a kind of variance of the gradients:

$$\frac{1}{n}\sum_{i=1}^{n}\|\underbrace{\nabla f_i(w^t)}_{\text{gradient of example 'i'}} - \underbrace{\nabla f(w^t)}_{\text{average gradient over all examples}}\|^2$$

Q: When is this variance zero?

Q: When is this variance large?

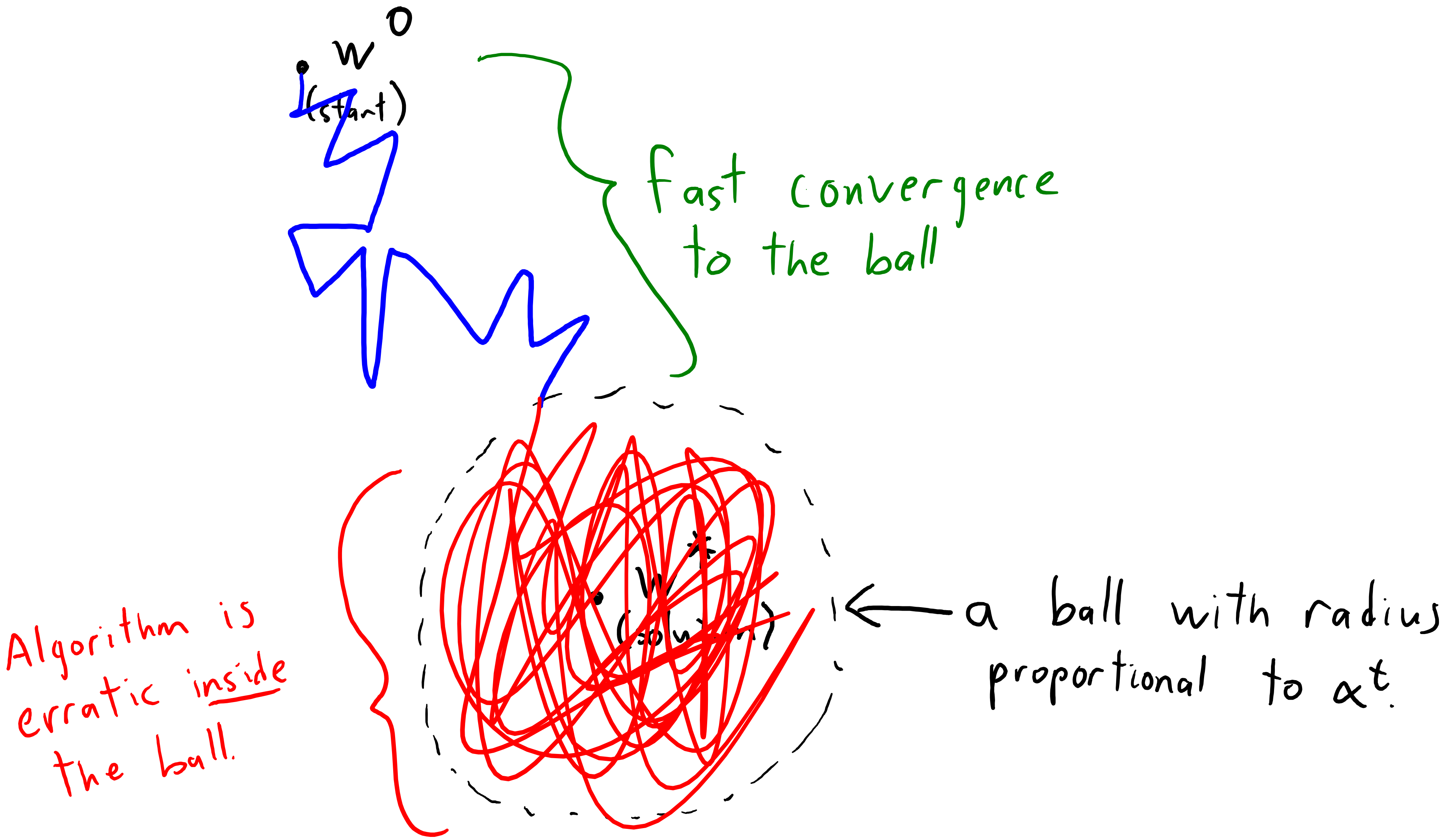
Effect of the Step-Size

- We can reduce the effect of the variance with the step size.
  - Variance slows progress by amount proportional to square of step-size.
  - So as the step size gets smaller, the variance has less of an effect.

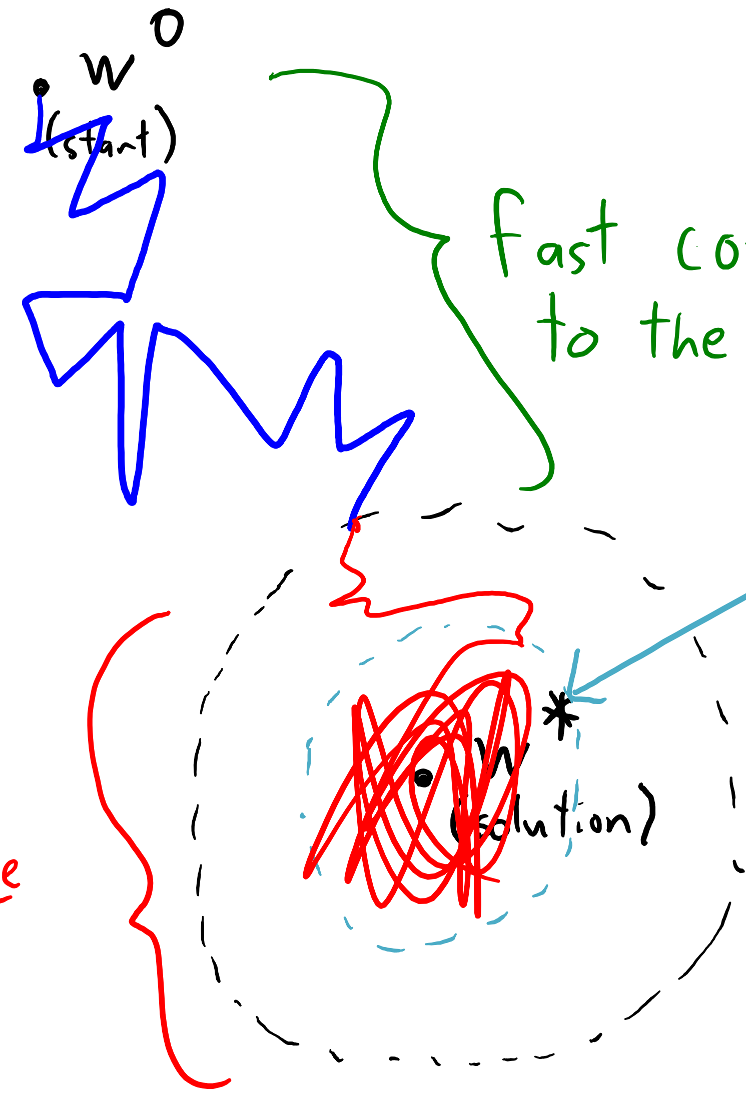
- For a fixed step-size, SG makes progress until variance is too big.

- This leads to two "phases" when we use a constant step-size:
  1. Rapid progress when we are (near/[✓ far from]) the solution.
  2. Erratic behaviour confined to a "ball" around solution.
     (Radius of ball is proportional to the step-size.)

Stochastic Gradient with Constant Step Size



Stochastic Gradient with Constant Step Size



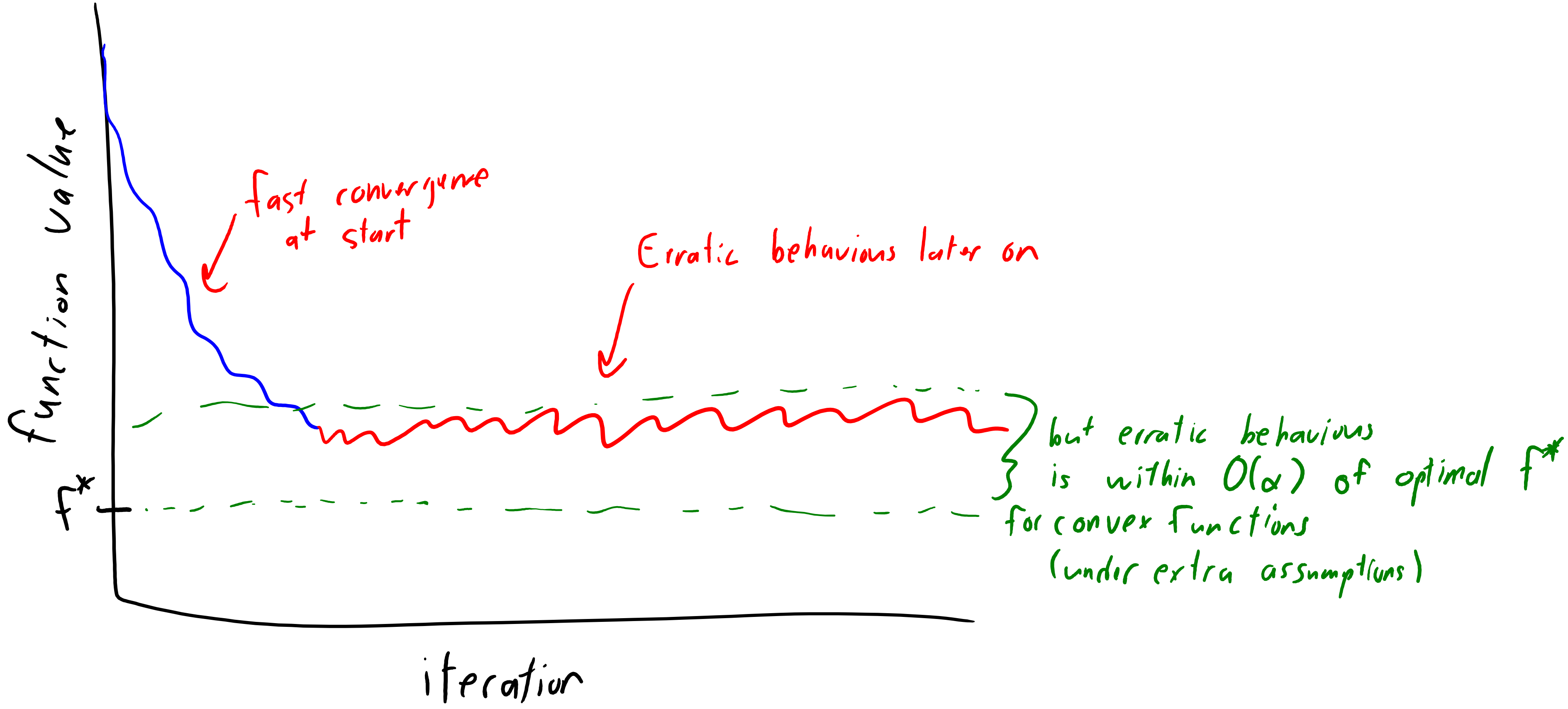
fast convergence to the ball

We can divide the radius of ball in 2 by dividing α^t by 2. (but takes longer to get to ball)

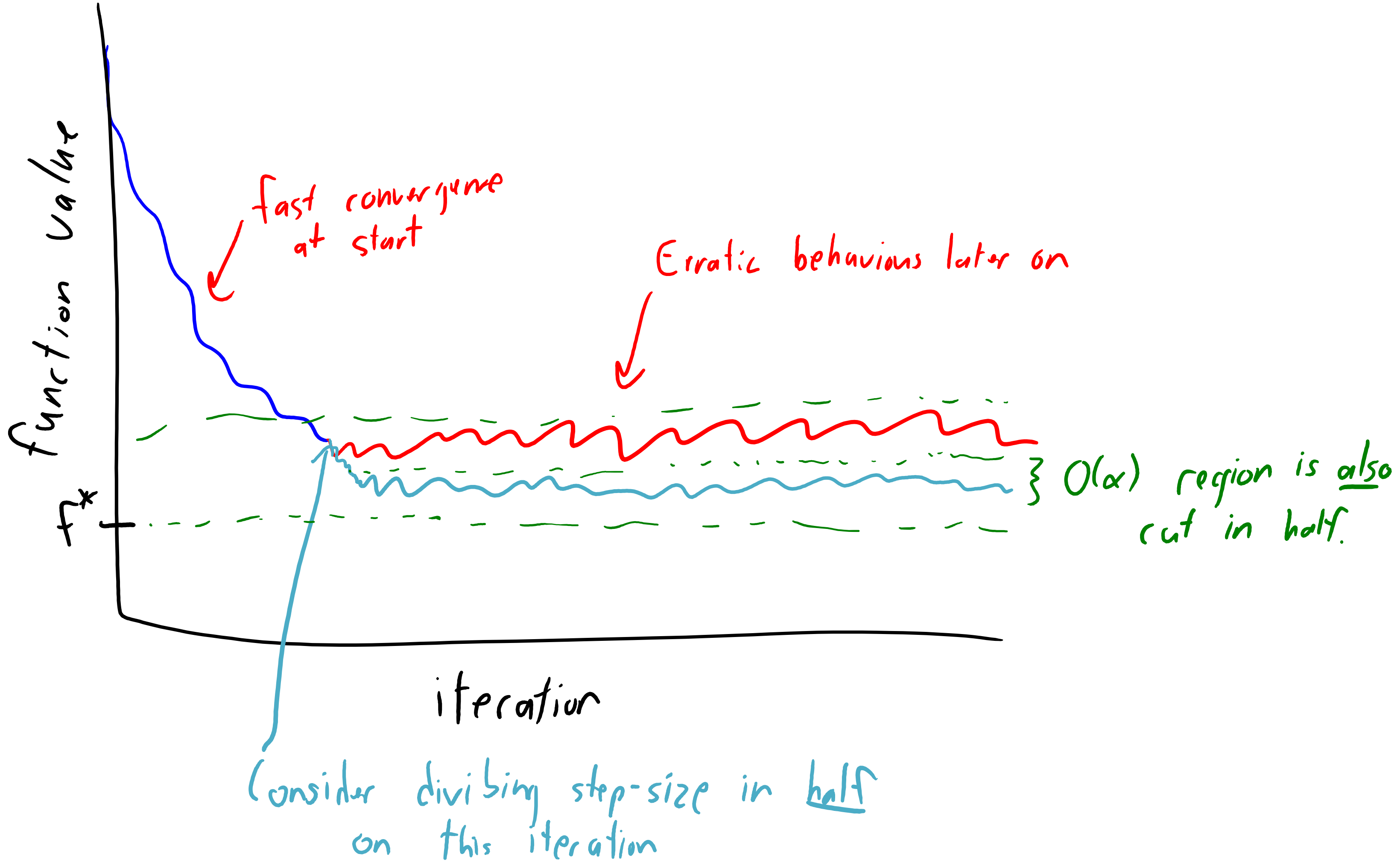
← a ball with radius proportional to α^t.

Algorithm is erratic <u>inside</u> the ball.

Stochastic Gradient with Constant Step Size



Stochastic Gradient with Constant Step Size



fast convergence at start

Erratic behaviour later on

} $O(\alpha)$ region is <u>also</u> cut in half.

Consider dividing step-size in <u>half</u> on this iteration

Step Size Considerations

α^t ← decay based on t.

- To get convergence, we need a decreasing step size.
  - Shrinks size of ball to zero so we converge to w^*.
- But it can’t shrink too quickly:
  - Otherwise, we don’t move fast enough to reach the ball.
- Stochastic gradient converges to a stationary point if:
  - “Total distance covered” grows faster than “squared L2-norm of step sequence”.

sensitivity to variance ↙

$\alpha^T\alpha$

L2-norm of $[\alpha^0, \alpha^1, \cdots,]$

$$\frac{\sum_{t=1}^{\infty}(\alpha^t)^2}{\sum_{t=1}^{\infty}\alpha^t} = 0$$

Integral of displacements
= total distance

  - This choice also works for non-smooth functions like SVMs.
    - Function must be continuous and not “too crazy”
      (we’re still figuring it out for non-convex).

Stochastic Gradient with Decreasing Step Sizes

- For convergence, step-sizes need to satisfy: $\sum_{t=1}^{\infty}(\alpha^t)^2 / \sum_{t=1}^{\alpha}\alpha^t = 0$
- Classic solution is to use a step-size sequence like $\alpha^t = O(1/t)$.

$$\underbrace{\sum_{t=1}^{\infty}\alpha^t = \sum_{t=1}^{\infty}\frac{1}{t} = \infty}_{\text{total distance covered by stepping forever}} \qquad \sum_{t=1}^{\infty}(\alpha^t)^2 = \underbrace{\sum_{t=1}^{\infty}\frac{1}{t^2} < \infty}_{\|\alpha\|^2,\ \alpha=[\alpha^0,\alpha^1,\dots]}$$

$$\sum \frac{\gamma}{t+c} \to \infty \qquad \sum \frac{\gamma}{t^2+c} \to < \infty$$

at some point, new step does not increase the norm.

  - E.g., $\alpha^t = .001/t$.
- Unfortunately, this often works badly in practice:
  - Steps get too small too fast.
  - Some authors add extra parameters like $\alpha^t = \gamma/(\beta t + \Delta)$, which helps a bit.
  - One of the only cases where this works well: binary SVMs with $\alpha^t = 1/\lambda t$.

Stochastic Gradient with Decreasing Step Sizes

- How do we pick step-sizes satisfying $\sum_{t=1}^{\infty}(\alpha^t)^2 / \sum_{t=1}^{\infty}\alpha^t = 0$

- Better solution is to use a step-size sequence like $\alpha^t = O(1/\sqrt{t})$.

$$\sum_{t=1}^{K}\alpha^t = \sum_{t=1}^{K}\frac{1}{\sqrt{t}} = O(\sqrt{K}) \qquad \sum_{t=1}^{k}(\alpha^t)^2 = \sum_{t=1}^{k}\frac{1}{t} = O(\log k)$$

  - E.g., use $\alpha^t = \underline{.001}/\sqrt{t} = \frac{c}{\sqrt{t}}$ ← hyperparameter of step size
  - Both sequences diverge, but denominator diverges faster.

- Roughly optimizes rate at which ratio goes to zero.
  - Better worst-case theoretical properties (and more robust to step-size).
  - Often better in practice too.

Stochastic Gradient with Constant Step Sizes?

- Alternately, could we just use a constant step-size?
  - E.g., use α^t = .001 for all ‘t’.

- This will not converge to a stationary point in general.
  - However, do we need it to converge?

- What if you only care about the first 2-3 digits of the test error?
  - Who cares if you aren’t able to get 10 digits of optimization accuracy?

- There is a step-size small enough to achieve any fixed accuracy.
  - Just need radius of “ball” to be small enough.

- Magic number: set α^t = 3e-4 (0.0003)

Mini-batches: Using more than 1 example

- Does it make sense to use more than 1 random example?
  - Yes, you can use a "mini-batch" B^t of examples. $(X, y) \rightarrow (X_{B^t}, y_{B^t}) \rightarrow \nabla f_{B^t}(w^t)$

$$w^{t+1} = w^t - \alpha^t \frac{1}{|B^t|} \sum_{i \in B^t} \nabla f_i(w^t)$$

Random "batch" of examples.

  - Radius of ball is inversely proportional to the mini-batch size.
    - If you double the batch size, you half the radius of the ball.
      - Big gains for going from 1 to 2, less gains from going from 100 to 101.
    - You can use a bigger step size as the batch size increases ("linear scaling" rule).
      - Gets you to the ball faster (though diverges if step-size gets too big).
  - Useful for vectorizing/parallelizing code.
    - Evaluate one gradient on each core.

A Practical Strategy for Deciding When to Stop

- In gradient descent, we can stop when gradient is close to zero.

- In stochastic gradient:
  - Individual gradients don't necessarily go to zero.
  - We can't see full gradient, so we don't know when to stop.

- Practical trick:
  - Every 'k' iterations (for some large 'k'), measure validation set error.
  - Stop if the validation set error "isn't improving".
    - We don't check the gradient, since it takes a lot longer for the gradient to get small.
    - This "early stopping" can also reduce overfitting.
    - "Snapshotting": save model to disk each time, use latest/best-performing snapshot

Summary

- Kernels let us use similarity between objects, rather than features.
  - Allows some exponential- or infinite-sized feature sets.
  - Applies to distance-based and linear models with L2-regularization.

- Stochastic gradient methods let us use huge datasets.
- Step-size in stochastic gradient is a huge pain:
  - Needs to go to zero to get convergence, but classic O(1/t) steps are bad.
  - O($1/\sqrt{t}$) works better, but still pretty slow.
  - Constant step-size is fast, but only up to a certain point.
- SGD practical issues: mini-batching, averaging, termination.
- SAG and other methods fix SG convergence for finite datasets. (bonus)
- Infinite datasets can be used with SG and do not overfit. (bonus)

- Next time: Using Probability for Machine Learning

Review Questions

- Q1: How does polynomial and Gaussian RBF kernels affect the shape of decision boundaries in linear classifiers?

- Q2: Can stochastic gradient descent help us with memory constraints?

- Q3: Why can stochastic gradient descent make progress even based on a single example?

- Q4: In what situation “early stopping” is a good idea? When is it a bad idea?

Kernel Trick for Other Methods

- Besides L2-regularized least squares, when can we use kernels?
  - "Representer theorems" (bonus slide) have shown that any L2-regularized linear model can be kernelized:

If learning can be written in the form $\min_v f(Zv) + \frac{1}{2}\underbrace{\|v\|^2}_{v^Tv}$ for some 'Z'

then under weak conditions ("representer theorem") we can re-parameterize in terms of $v = Z^Tu$

giving $$\min_u f(\underbrace{Z\overbrace{Z^Tu}^{v}}_{K}) + \frac{1}{2}\overbrace{u^TZ}^{v^T}\underbrace{\overbrace{Z^Tu}^{v}}_{K}$$

Only need 'K'

At test time you would use $\tilde{Z}\underbrace{v}_{Z^Tu} = \underbrace{\tilde{Z}Z^T}_{\tilde{K}}u = \tilde{K}u$

Kernel Trick for Other Methods

- Besides L2-regularized least squares, when can we use kernels?
  - "Representer theorems" (bonus slide) have shown that any L2-regularized linear model can be kernelized:
    - L2-regularized robust regression.
    - L2-regularized brittle regression.
    - L2-regularized logistic regression.
    - L2-regularized hinge loss (SVMs).

With a particular implementation, can reduce prediction cost from $O(ndt)$ to $O(mdt)$.

Number of support vectors.

Kernel Trick for Non-Vector Data

- Consider data that doesn’t look like this:

$$X = \begin{bmatrix} 0.5377 & 0.3188 & 3.5784 \\ 1.8339 & -1.3077 & 2.7694 \\ -2.2588 & -0.4336 & -1.3499 \\ 0.8622 & 0.3426 & 3.0349 \end{bmatrix}, \quad y = \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix},$$

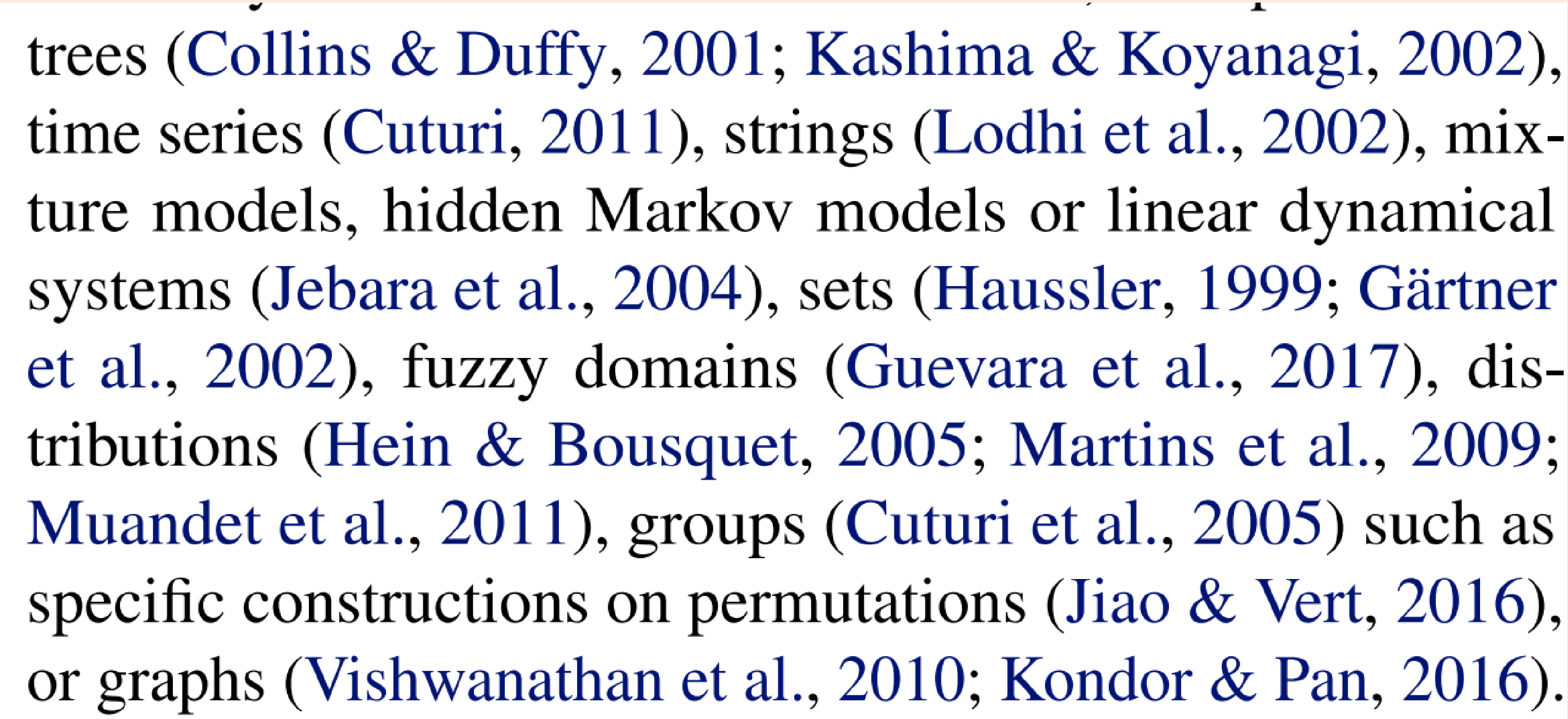
- But instead looks like this:

$$X = \begin{bmatrix} \text{Do you want to go for a drink sometime?} \\ \text{J'achète du pain tous les jours.} \\ \text{Fais ce que tu veux.} \\ \text{There are inner products between sentences?} \end{bmatrix}, y = \begin{bmatrix} +1 \\ -1 \\ -1 \\ +1 \end{bmatrix}.$$

- We can interpret $k(x_i,x_j)$ as a “similarity” between objects xi and xj.
  - We don’t need features if we can compute “similarity” between objects.
  - Kernel trick lets us fit regression models without explicit features.
  - There are “string kernels”, “image kernels”, “graph kernels”, and so on.

Kernel Trick for Non-Vector Data

- Recent list of types of data where people have defined kernels:



trees (Collins & Duffy, 2001; Kashima & Koyanagi, 2002), time series (Cuturi, 2011), strings (Lodhi et al., 2002), mixture models, hidden Markov models or linear dynamical systems (Jebara et al., 2004), sets (Haussler, 1999; Gärtner et al., 2002), fuzzy domains (Guevara et al., 2017), distributions (Hein & Bousquet, 2005; Martins et al., 2009; Muandet et al., 2011), groups (Cuturi et al., 2005) such as specific constructions on permutations (Jiao & Vert, 2016), or graphs (Vishwanathan et al., 2010; Kondor & Pan, 2016).

- Bonus slide overviews a particular “string” kernel.

https://arxiv.org/pdf/1802.04784.pdf

Valid Kernels

- What kernel functions $k(x_i,x_j)$ can we use?

- Kernel 'k' must be an inner product in some space:
  - There must exist a mapping from the x_i to some z_i such that $k(x_i,x_j) = z_i^T z_j$.

- It can be hard to show that a function satisfies this.
  - Infinite-dimensional eigenfunction problem.

- But like convex functions, there are some simple rules for constructing "valid" kernels from other valid kernels (bonus slide).

Polyak-Ruppert Iterate Averaging

- Another practical/theoretical trick is averaging of the iterations.
  1. Run the stochastic gradient algorithm with $\alpha^t = O(1/\sqrt{t})$ or α^t constant.
  2. Take some weighted average of the w^t values.

$$\bar{w}^t = \sum_{k=1}^{t} v^k w^k$$

Here, v^k is a scalar "weight" of iteration 'k'

Uniform average:

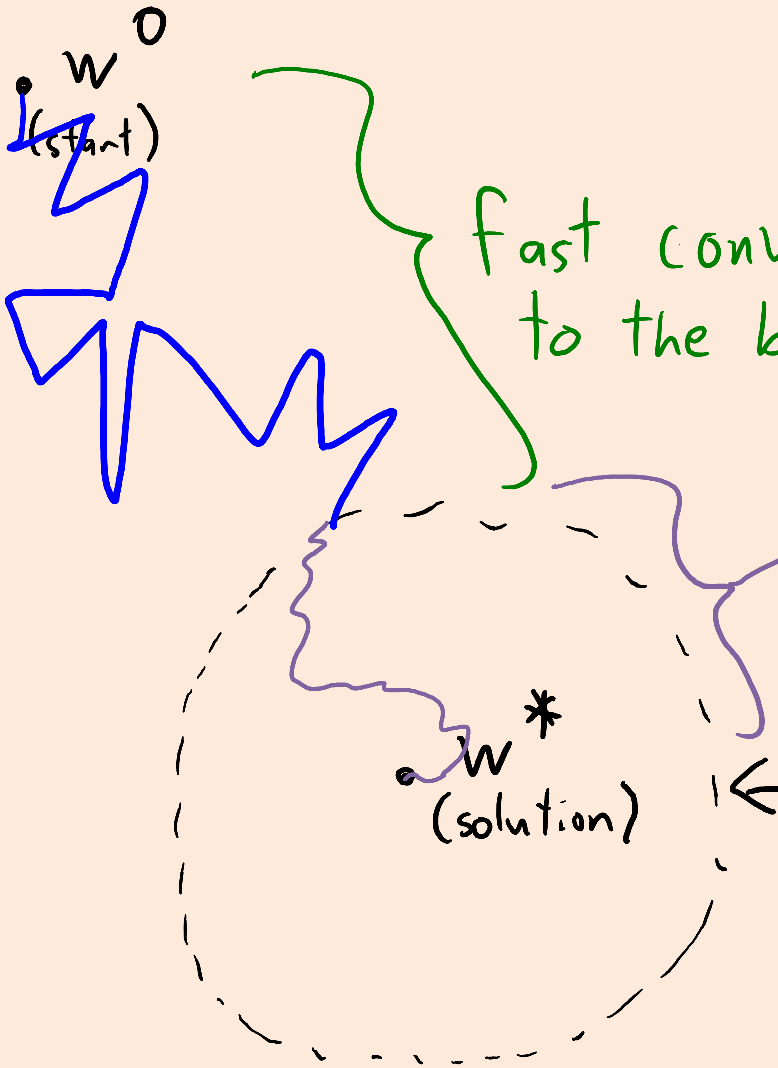
$$\bar{w}^t = \underbrace{\frac{1}{t}}_{v^k = \frac{1}{t}} \sum_{k=1}^{t} w^k$$

- Average does not affect the algorithm, it's just "watching".
- Surprising result shown by Polyak and by Ruppert in the 1980s:
  - Asymptotically converges as fast as stochastic Newton's method.

Stochastic Gradient with Averaging

Often, you average the second half of the iterations.

Set $\bar{w}^t = \frac{1}{t/2} \sum_{k=t/2}^{t} w^k$

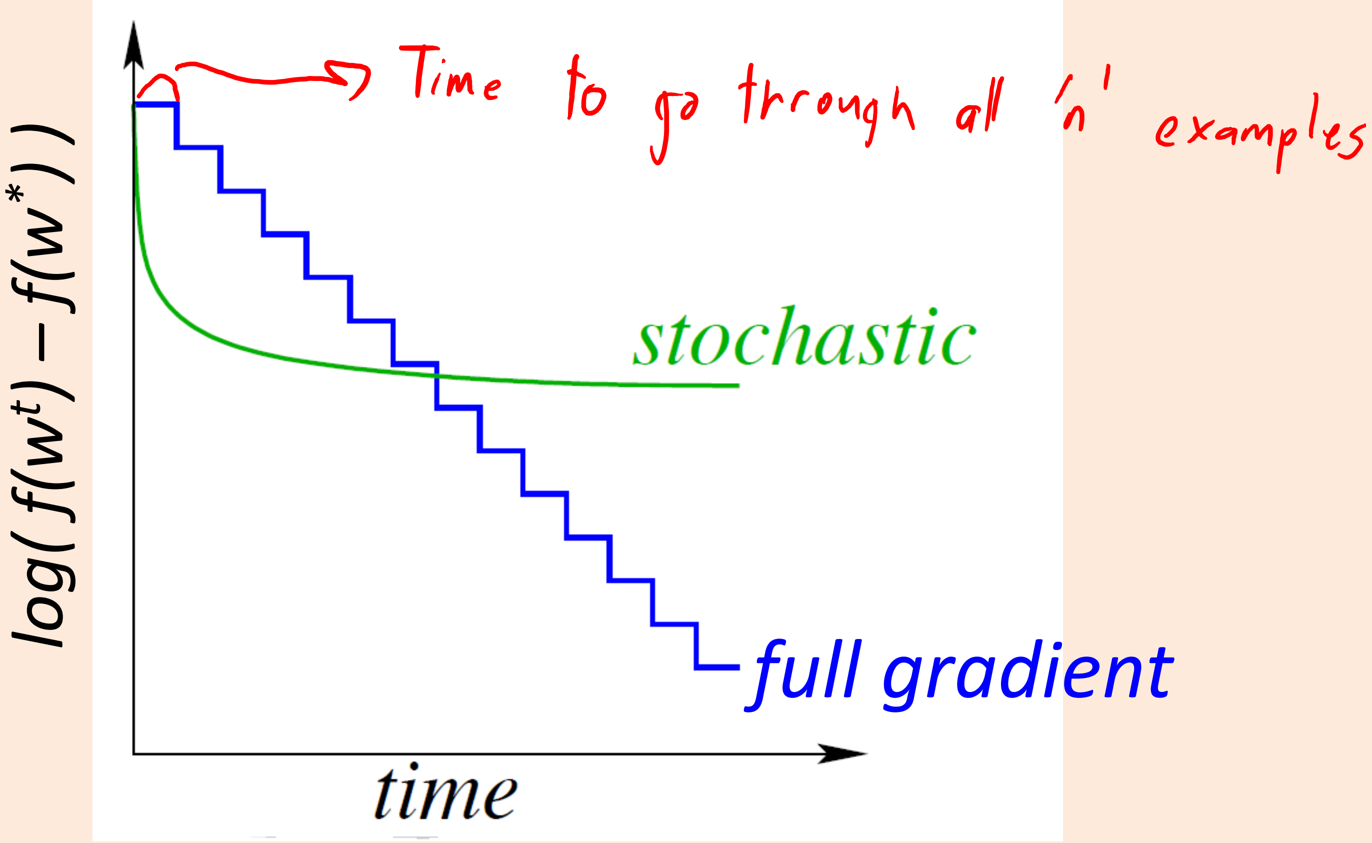


fast convergence to the ball

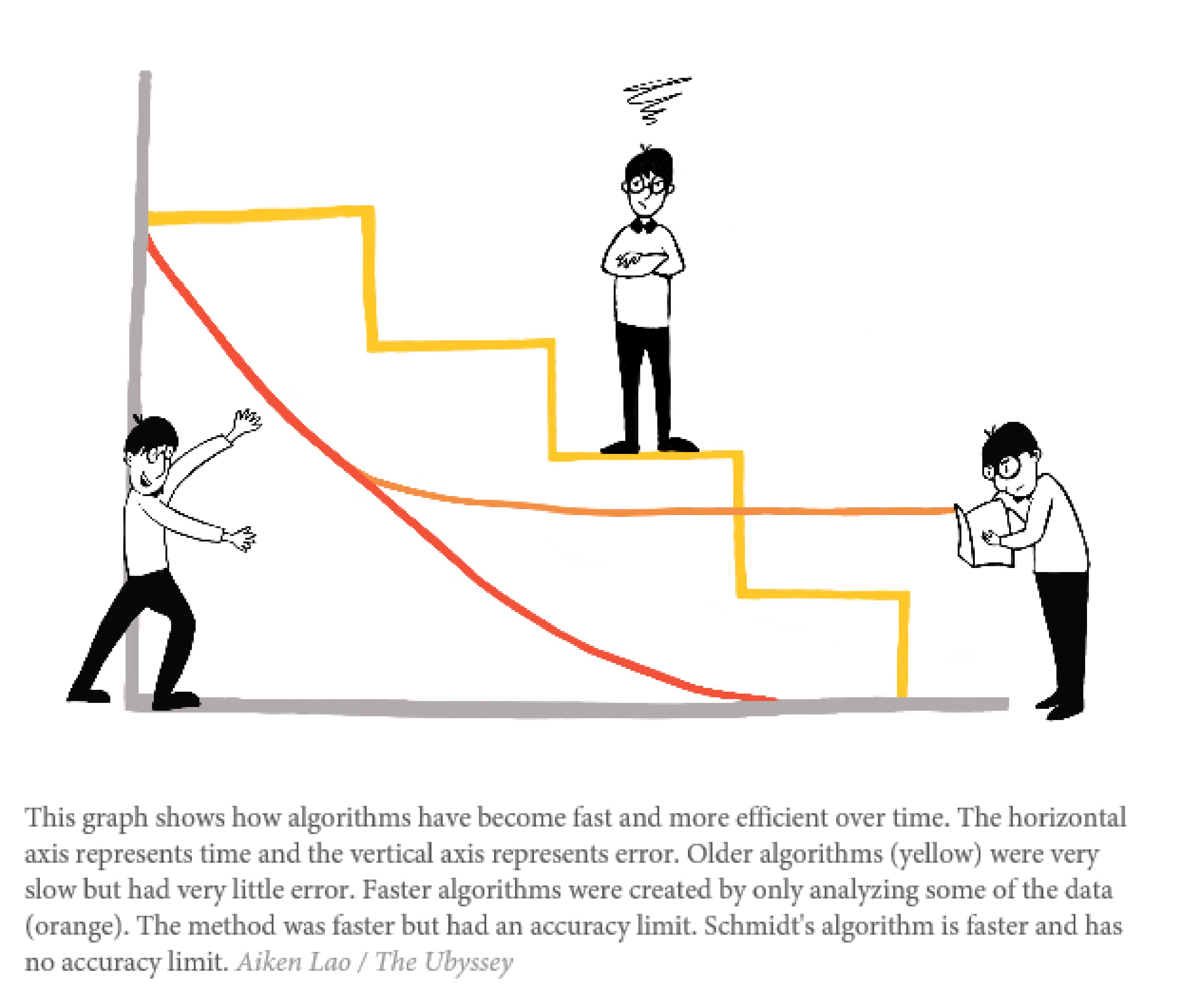
Average of erratic behaviour can converge to w^*.

a ball with radius proportional to α^t.

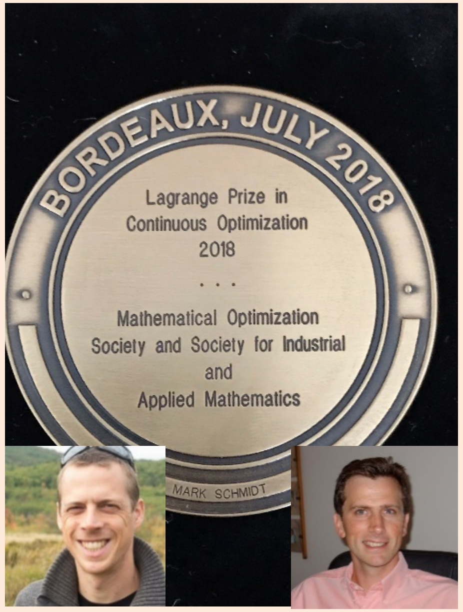
Gradient Descent vs. Stochastic Gradient



- 2012: methods with cost of stochastic gradient, progress of full gradient.
  - Key idea: if ‘n’ is finite, you can use a memory instead of having α_t go to zero.
  - First was stochastic average gradient (SAG), “low-memory” version is SVRG.



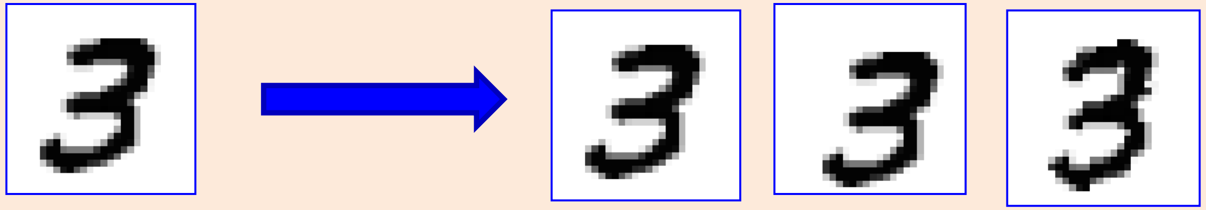
This graph shows how algorithms have become fast and more efficient over time. The horizontal axis represents time and the vertical axis represents error. Older algorithms (yellow) were very slow but had very little error. Faster algorithms were created by only analyzing some of the data (orange). The method was faster but had an accuracy limit. Schmidt's algorithm is faster and has no accuracy limit. *Aiken Lao / The Ubyssey*


BORDEAUX, JULY 2018
Lagrange Prize in Continuous Optimization 2018
. . .
Mathematical Optimization Society and Society for Industrial and Applied Mathematics
MARK SCHMIDT

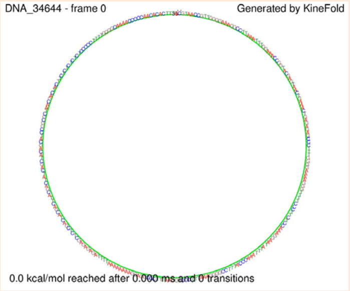
https://www.ubyssey.ca/science/schmidt-sloan-fellowship/

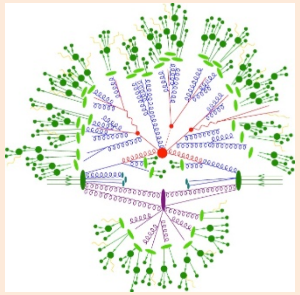
Machine Learning with "n = ∞"

- Here are some scenarios where you effectively have "n = ∞":
  - A dataset that is so large we cannot even go through it once (Gmail).
  - A function you want to minimize that you can't measure without noise.
  - You want to encourage invariance with a continuous set of transformation:
    - You consider infinite number of translations/rotations instead of a fixed number.



  - Learning from simulators with random numbers (physics/chem/bio):





http://kinefold.curie.fr/cgi-bin/form.pl
https://sciencenode.org/feature/sherpa-and-open-science-grid-predicting-emergence-jets.php

Stochastic Gradient with Infinite Data

- Previous slide gives examples with infinite sequence of IID samples.

- How can you practically train on infinite-sized datasets?

- Approach 1 (exact optimization on finite ‘n’):
  - Grab ‘n’ data points, for some really large ‘n’.
  - Fit a regularized model on this fixed dataset (“empirical risk minimization”).

- Approach 2 (stochastic gradient for ‘n’ iterations):
  - Run stochastic gradient iteration for ‘n’ iterations.
  - Each iteration considers a new example, never re-visiting any example.

Stochastic Gradient with Infinite Data

- Approach 2 works because of an amazing property of stochastic gradient:
  - The classic convergence analysis does not rely on 'n' being finite.
- Further Approach 2 only looks at a data point once:
  - Each example is an unbiased approximation of test data.
- So Approach 2 is doing stochastic gradient on test error:
  - It cannot overfit.
- Up to a constant, Approach 1 and 2 have same test error bound.
  - This is sometimes used to justify SG as the "ultimate" learning algorithm.
    - "Optimal test error by computing gradient of each example once!"
  - In practice, Approach 1 usually gives lower test error.
    - The constant factor matters!

A Practical Strategy For Choosing the Step-Size

- All these step-sizes have a constant factor in the "O" notation.
  - E.g.,

$$\alpha^t = \frac{\gamma}{\sqrt{t}} \leftarrow \text{How do we choose this constant?}$$

- We don't know how to set step size as we go in the stochastic case.
  - And choosing wrong γ can destroy performance.

- Common practical trick:
  - Take a small amount of data (maybe 5% of the original data).
  - Do a binary search for γ that most improves objective on this subset.