Optimization and (Under/over)fitting

EECS 442 – Prof. David Fouhey

Winter 2019, University of Michigan

http://web.eecs.umich.edu/~fouhey/teaching/EECS442_W19/

Administrivia

- We’re grading HW2 and will try to get it back ASAP
- Midterm next week
  - Review sessions next week (go to whichever you want)
  - We will post priorities from slides
  - There’s a post on piazza looking for topics you want reviewed

Previously

Regularized Least Squares

Add **regularization** to objective that prefers some solutions:

Before: $\arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$ → Loss

After: $\arg\min_{\boldsymbol{w}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2$

Loss | Trade-off | Regularization

Want model "smaller": pay a penalty for **w** with big norm

Intuitive Objective: accurate model (low loss) but not too complex (low regularization). λ controls how much of each.

Nearest Neighbors

Known Images
Labels

Test
Image

x_1 $D(x_1, x_T)$ x_T

$D(x_N, x_T)$

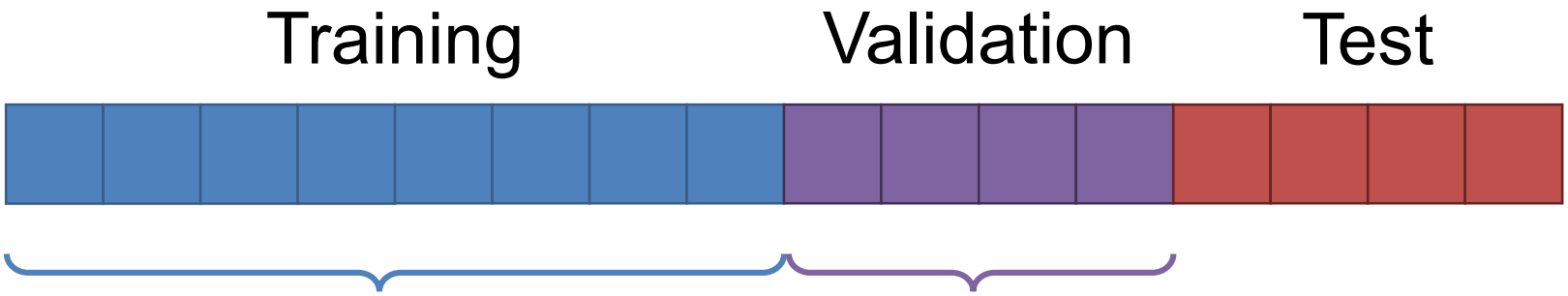
...

x_N

(1) Compute distance between feature vectors (2) find nearest (3) use label.

Picking Parameters

What distance? What value for k / λ?



Linear Models

Example Setup: 3 classes



Model – one weight per class: $\boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{w}_2$

Want:

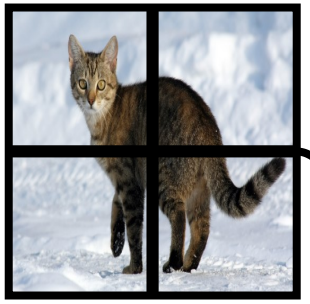
$\boldsymbol{w}_0^T \boldsymbol{x}$ big if cat

$\boldsymbol{w}_1^T \boldsymbol{x}$ big if dog

$\boldsymbol{w}_2^T \boldsymbol{x}$ big if hippo

Stack together: $\boldsymbol{W}_{\boldsymbol{3xF}}$ where **x** is in R^F

Linear Models



Cat weight vector	0.2	-0.5	0.1	2.0	1.1
Dog weight vector	1.5	1.3	2.1	0.0	3.2
Hippo weight vector	0.0	0.3	0.2	-0.3	-1.2

W

56
231
24
2
1

x_i

-96.8	Cat score
437.9	Dog score
61.95	Hippo score

Wx_i

Weight matrix a collection of scoring functions, one per class

Prediction is vector where jth component is “score” for jth class.

Diagram by: Karpathy, Fei-Fei

Objective 1: Multiclass SVM

Inference (x,y): $\arg\max_k (\boldsymbol{W}\boldsymbol{x})_k$

(Take the class whose weight vector gives the highest score)

Training ($\mathbf{x}_i$,y_i):

$$\arg\min_{\boldsymbol{W}} \lambda\|\boldsymbol{W}\|_2^2 + \sum_{i=1}^{n}\sum_{j\neq y_i} \max(0, (\boldsymbol{W}\boldsymbol{x}_i)_j - (\boldsymbol{W}\boldsymbol{x}_i)_{y_i} + m)$$

Regularization

Over all data points

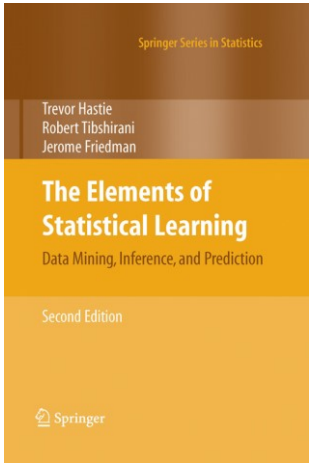
For every class j that's NOT the correct one (y_i)

Pay no penalty if prediction for class y_i is bigger than j by m ("margin"). Otherwise, pay proportional to the score of the wrong class.

Objective 1:

Called: Support Vector Machine

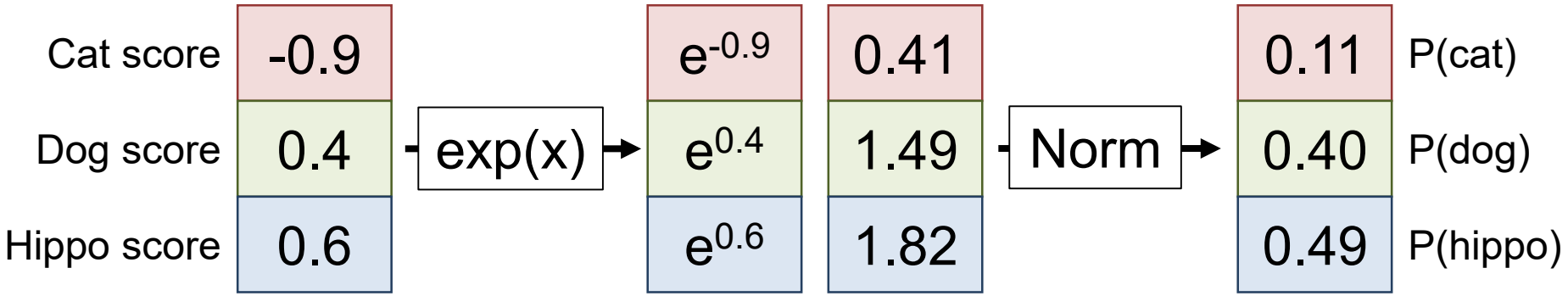
Lots of great theory as to why this is a sensible thing to do. See



Useful book (Free too!):
The Elements of Statistical Learning
Hastie, Tibshirani, Friedman
https://web.stanford.edu/~hastie/ElemStatLearn/

Objective 2: Making Probabilities

Converting Scores to "Probability Distribution"



$\sum$=3.72

Generally P(class j): $\frac{\exp((Wx)_j)}{\sum_k \exp((Wx)_k)}$

Called softmax function

Inspired by: Karpathy, Fei-Fei

Objective 2: Softmax

Inference (x): $\arg\max_{k} \ (\boldsymbol{W}\boldsymbol{x})_k$

(Take the class whose weight vector gives the highest score)

P(class j): $\dfrac{\exp((Wx)_j)}{\sum_k \exp((Wx)_k)}$

Why can we skip the exp/sum exp thing to make a decision?

Objective 2: Softmax

Inference (x): $\arg\max_k (\boldsymbol{W}\boldsymbol{x})_k$

(Take the class whose weight vector gives the highest score)

Training ($\mathbf{x}_i$,y_i):

$$\arg\min_{\boldsymbol{W}} \boldsymbol{\lambda}\|\boldsymbol{W}\|_2^2 + \sum_{i=1}^{n} -\log\left(\frac{\exp((Wx)_{y_i})}{\sum_k \exp((Wx)_k)}\right)$$

P(correct class)

Regularization

Over all data points

Pay penalty for not making correct class likely.
"Negative log-likelihood"

Today

- Optimizing things
- Time permitting:
  - (Under/over)fitting or (Bias/Variance)
  - (Not on the exam)

How Do We Optimize Things?

Goal: find the **w** minimizing some loss function L.

$$\arg\min_{\boldsymbol{w}\in R^N} L(\boldsymbol{w})$$

Works for lots of different Ls:

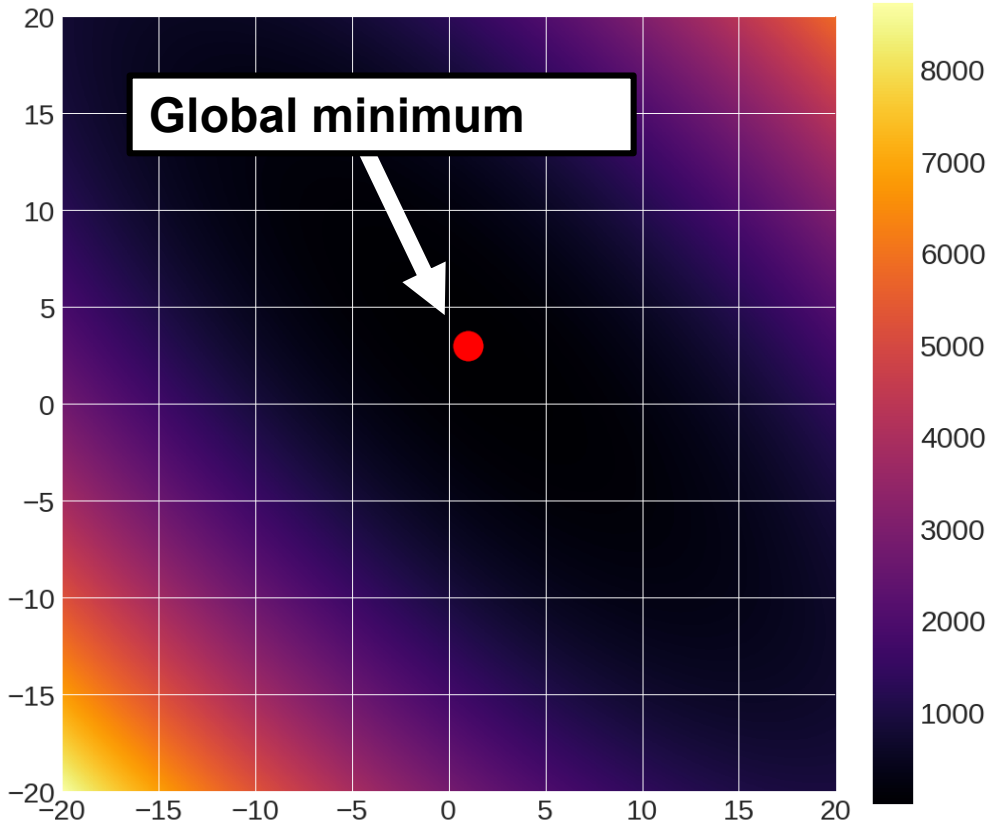
$$L(\boldsymbol{W}) = \boldsymbol{\lambda}\|\boldsymbol{W}\|_{\boldsymbol{2}}^{\boldsymbol{2}} + \sum_{i=1}^{n} -\log\left(\frac{\exp((Wx)_{y_i})}{\sum_k \exp((Wx)_k))}\right)$$

$$L(\boldsymbol{w}) = \lambda\|\boldsymbol{w}\|_2^2 + \sum_{i=1}^{n}\left(y_i - \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}}\right)^2$$

$$L(\boldsymbol{w}) = C\|\boldsymbol{w}\|_2^2 + \sum_{i=1}^{n}\max(0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_{\boldsymbol{i}})$$

Sample Function to Optimize

$$f(x,y) = (x+2y-7)^2 + (2x+y-5)^2$$

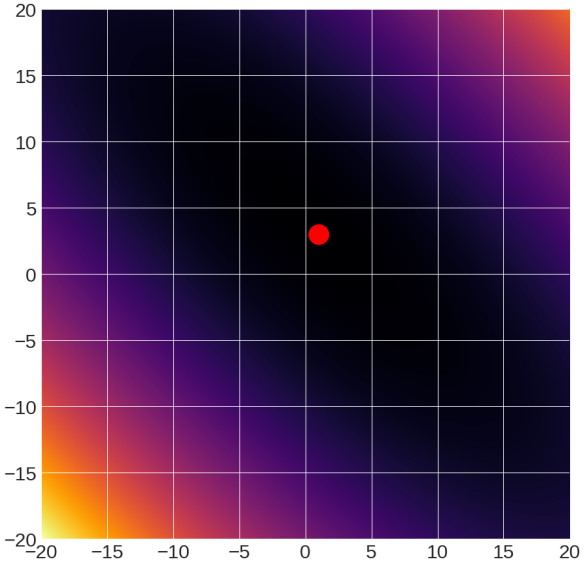


Sample Function to Optimize

- I'll switch back and forth between this 2D function (called the *Booth Function*) and other more-learning-focused functions
- Beauty of optimization is that it's all the same in principle.
- But don't draw too many conclusions: 2D space has qualitative differences from 1000D space.

See intro of: Dauphin et al. *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization* NIPS 2014
https://ganguli-gang.stanford.edu/pdf/14.SaddlePoint.NIPS.pdf

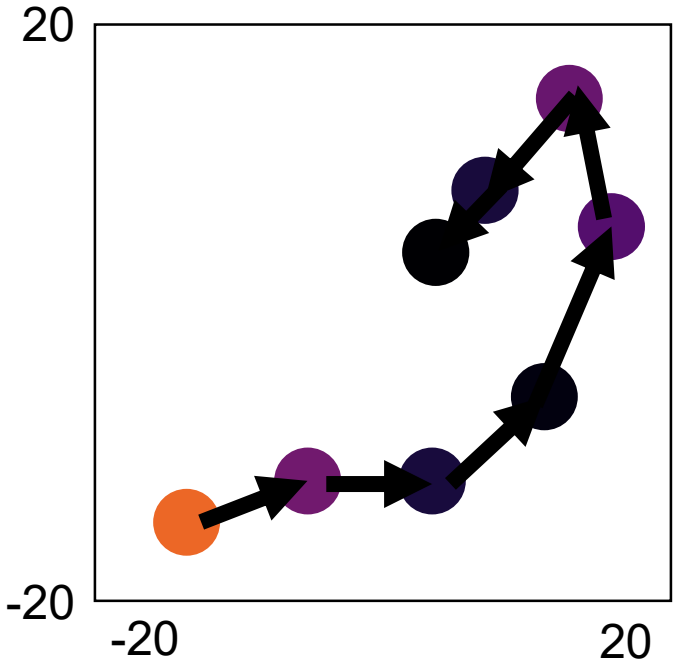
A Caveat

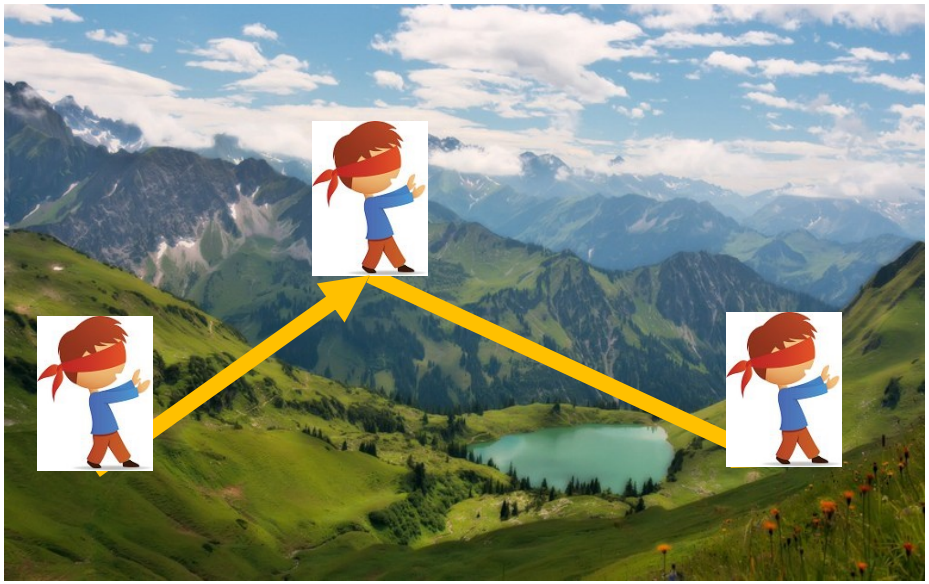


- Each point in the picture is a function evaluation
- Here it takes microseconds – so we can easily see the answer
- Functions we want to optimize may take hours to evaluate

A Few Caveat

Model in your head: moving around a landscape with a teleportation device



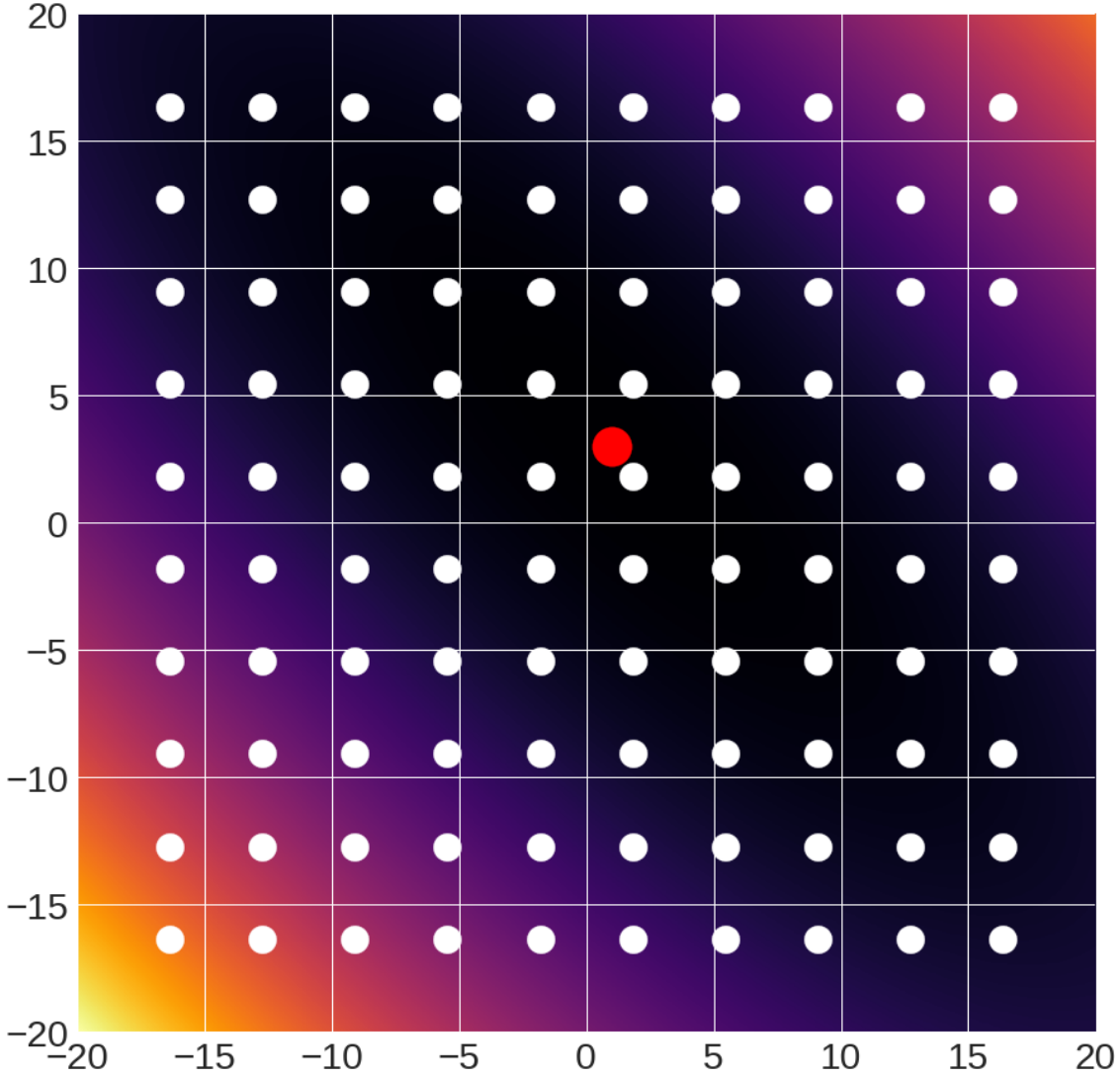


Landscape diagram: Karpathy and Fei-Fei

Option #1A – *Grid Search*

```
#systematically try things
best, bestScore = None, Inf
for dim1Value in dim1Values:
        ….
        for dimNValue in dimNValues:
                w = [dim1Value, …, dimNValue]
                if L(w) < bestScore:
                        best, bestScore = w, L(w)
return best
```

Option #1A – *Grid Search*



Option #1A – “Grid Search”

Pros:

1. Super simple
2. Only requires being able to evaluate model

Cons:

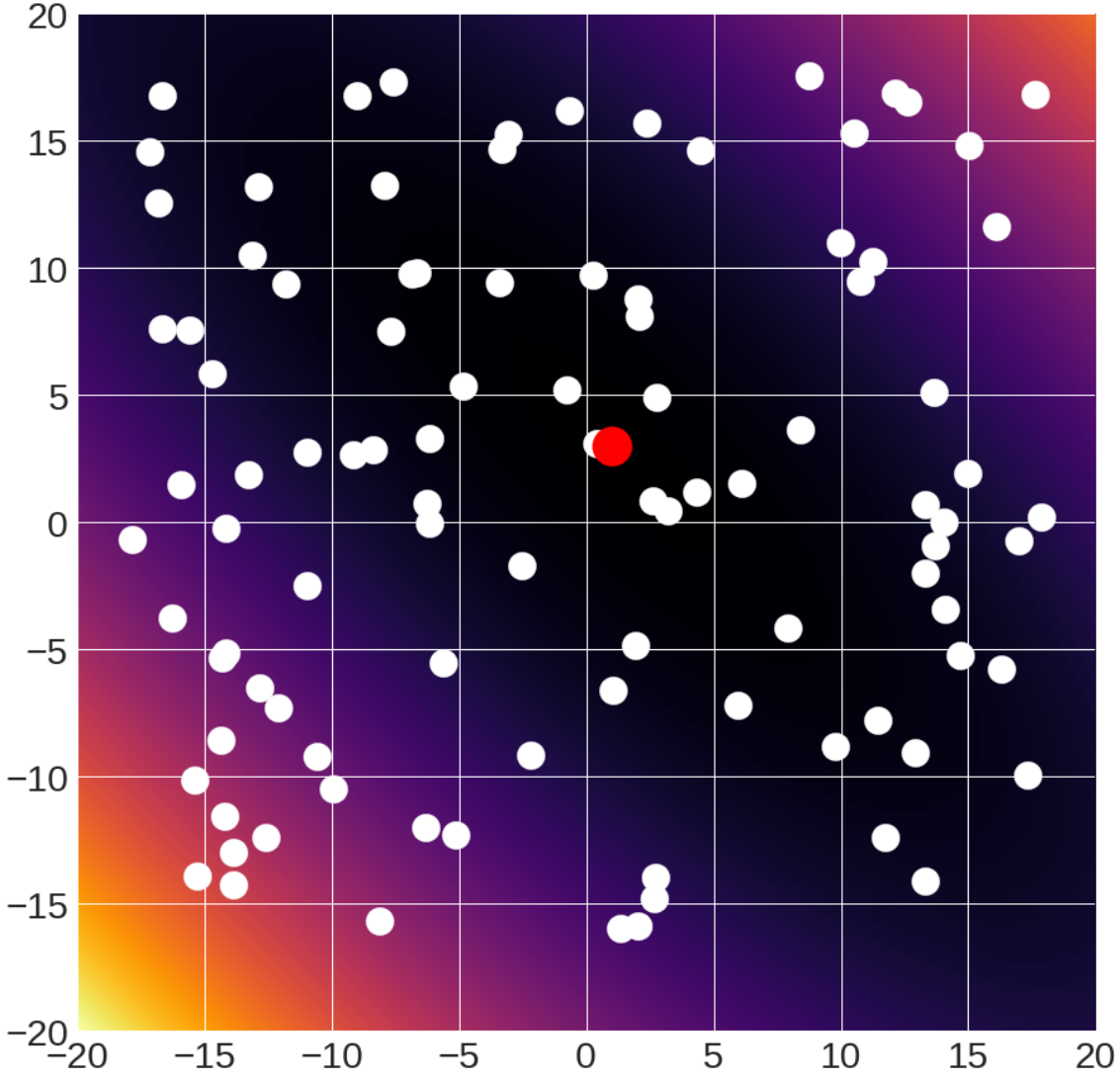
1. Scales horribly to high dimensional spaces

Complexity: $\text{samplesPerDim}^{\text{numberOfDims}}$

Option #1B – Random Search

#Do random stuff RANSAC Style
best, bestScore = None, Inf
for iter in range(numIters):
    w = random(N,1) #sample
    score = $L(\boldsymbol{w})$ #evaluate
    if score < bestScore:
        best, bestScore = **w**, score
return best

Option #1B – Random Search



Option #1B – Random Search

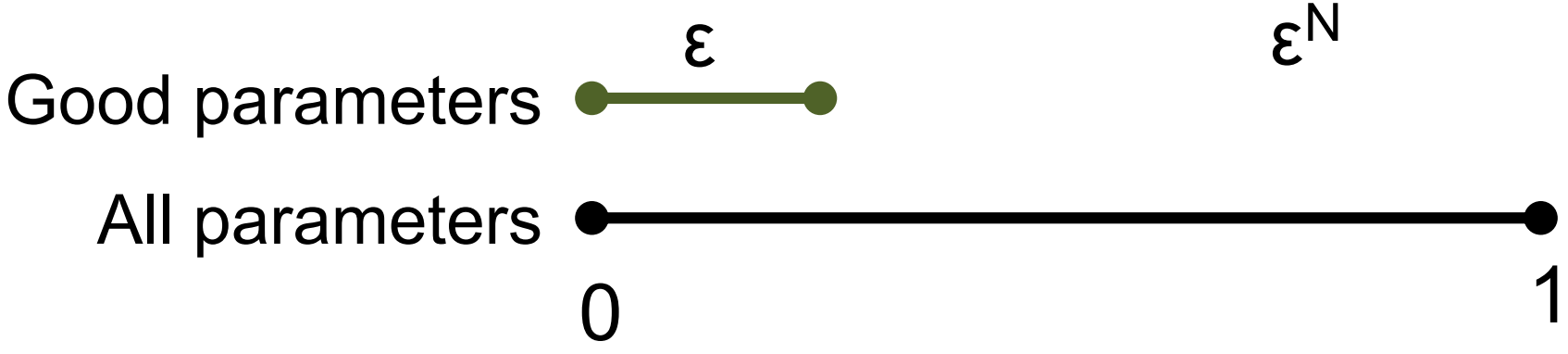
Pros:

1. Super simple
2. Only requires being able to sample model and evaluate it

Cons:

1. Slow and stupid – throwing darts at high dimensional dart board
2. Might miss something

$P(\text{all correct}) = \varepsilon^N$



When Do You Use Options 1A/1B?

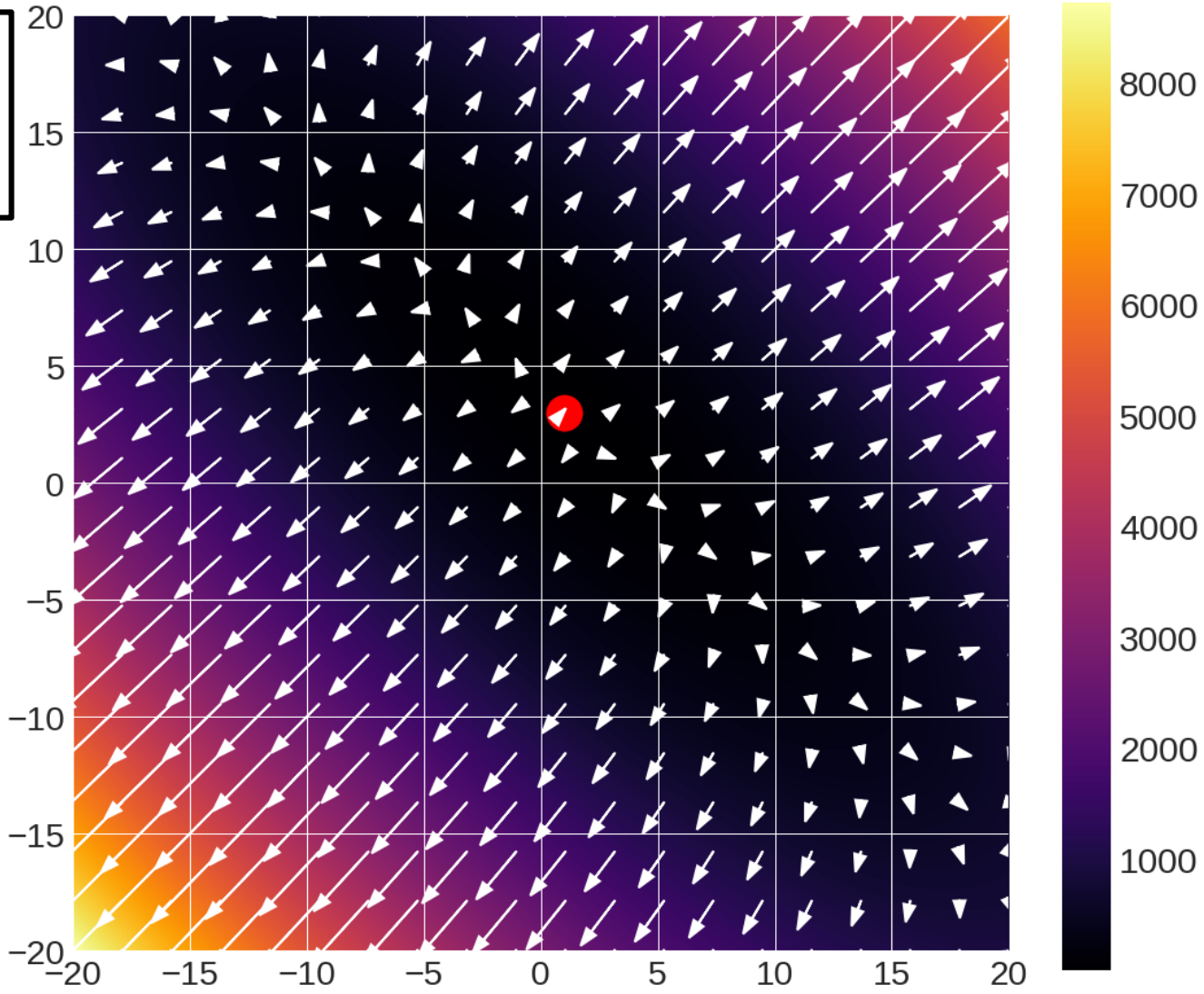
Use these when

- Number of dimensions small, space bounded
- Objective is impossible to analyze (e.g., test accuracy if we use this distance function)

Random search is arguably more effective; grid search makes it easy to systematically test something (people love certainty)

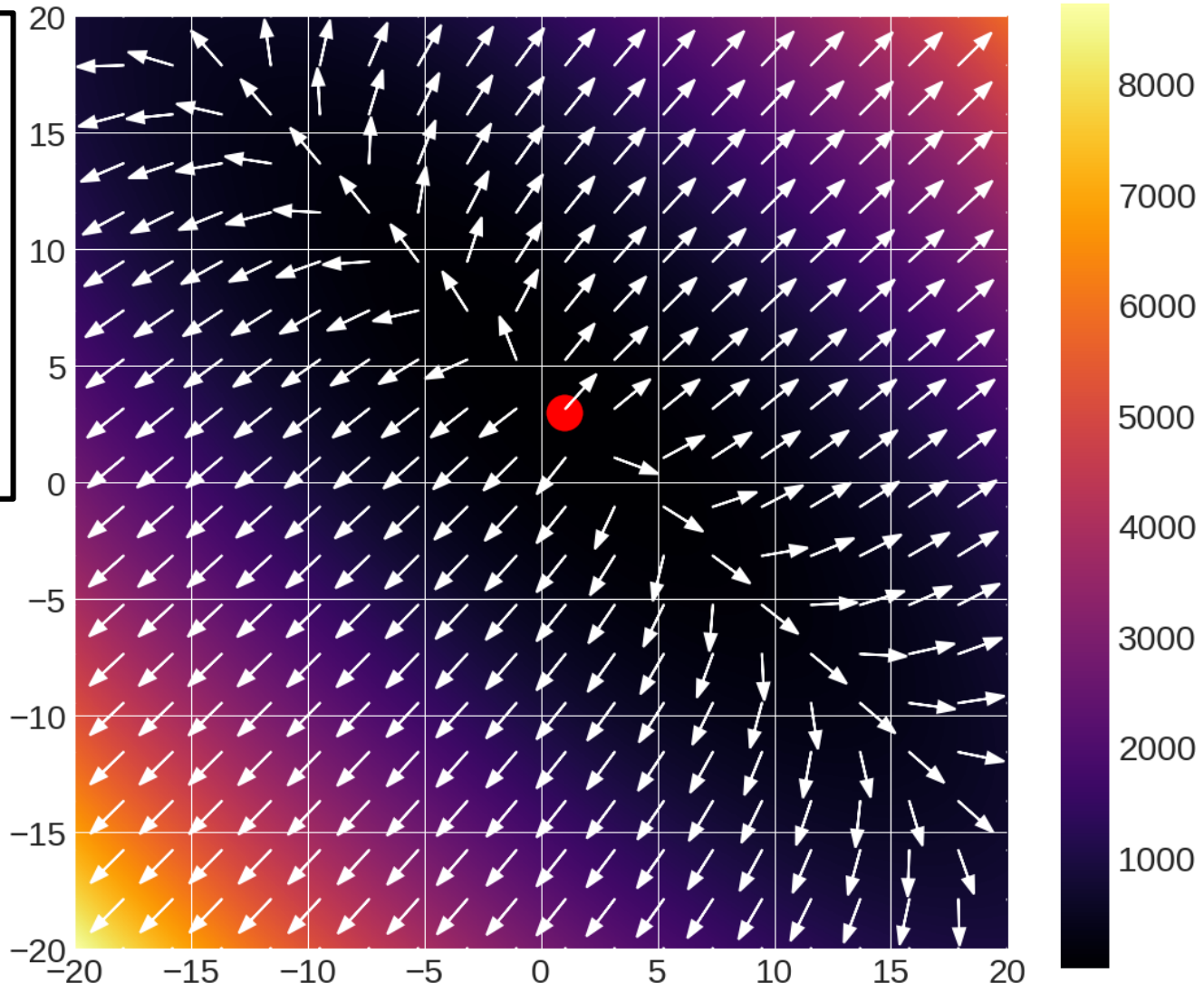
Option 2 – Use The Gradient

Arrows:
gradient



Option 2 – Use The Gradient

Arrows: **gradient direction** (scaled to unit length)



Option 2 – Use The Gradient

Want: $\arg\min_{\boldsymbol{w}} L(\boldsymbol{w})$

What's the geometric interpretation of:

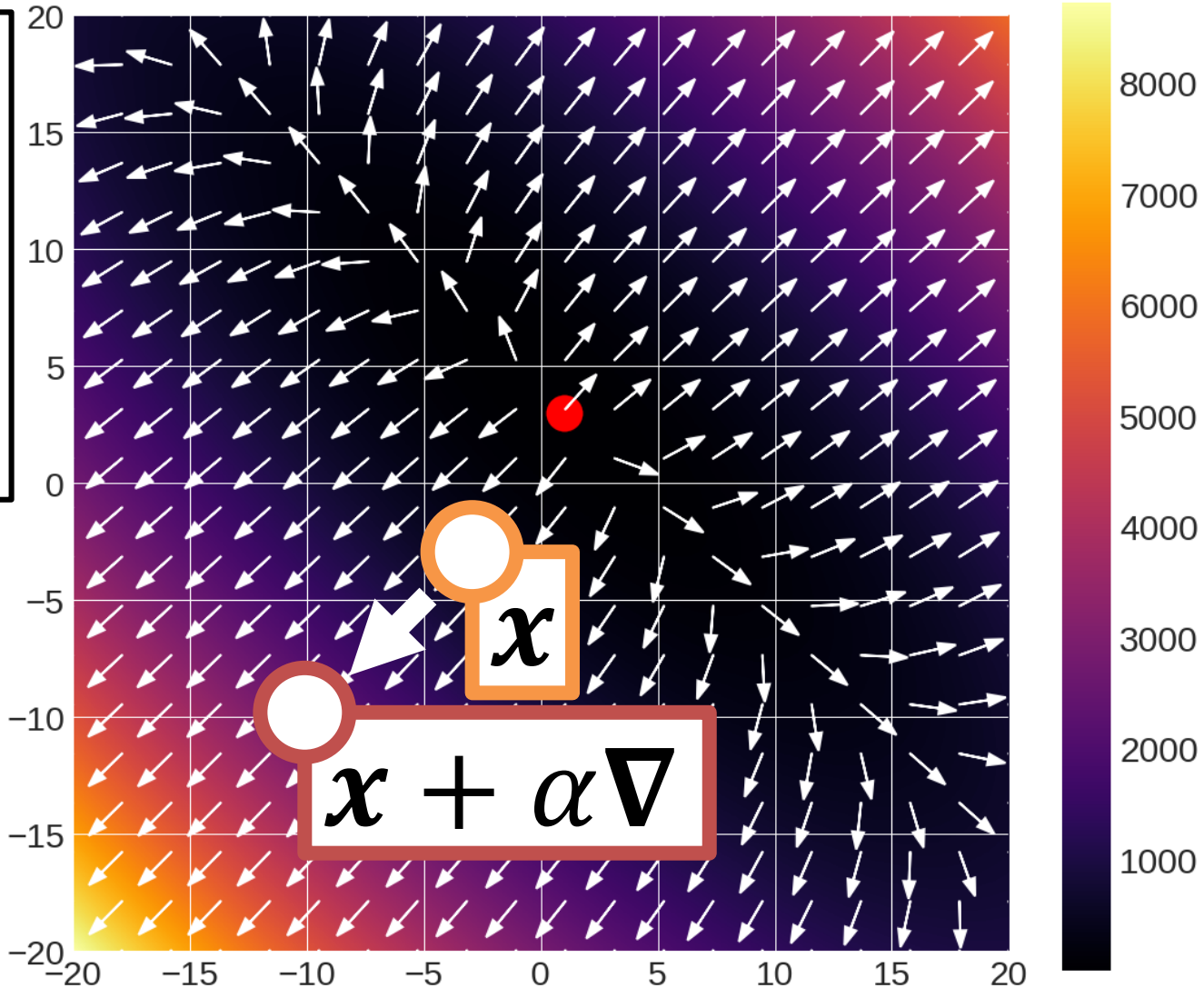
$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \begin{bmatrix} \partial L/\partial \boldsymbol{x}_1 \\ \vdots \\ \partial L/\partial \boldsymbol{x}_N \end{bmatrix}$$

Which is bigger (for small α)?

$$L(\boldsymbol{x}) \quad \begin{matrix} \leq ? \\ \\ > ? \end{matrix} \quad L(\boldsymbol{x} + \alpha \nabla_{\boldsymbol{w}} L(\boldsymbol{w}))$$

Option 2 – Use The Gradient

Arrows: gradient direction (scaled to unit length)



Option 2 – Use The Gradient

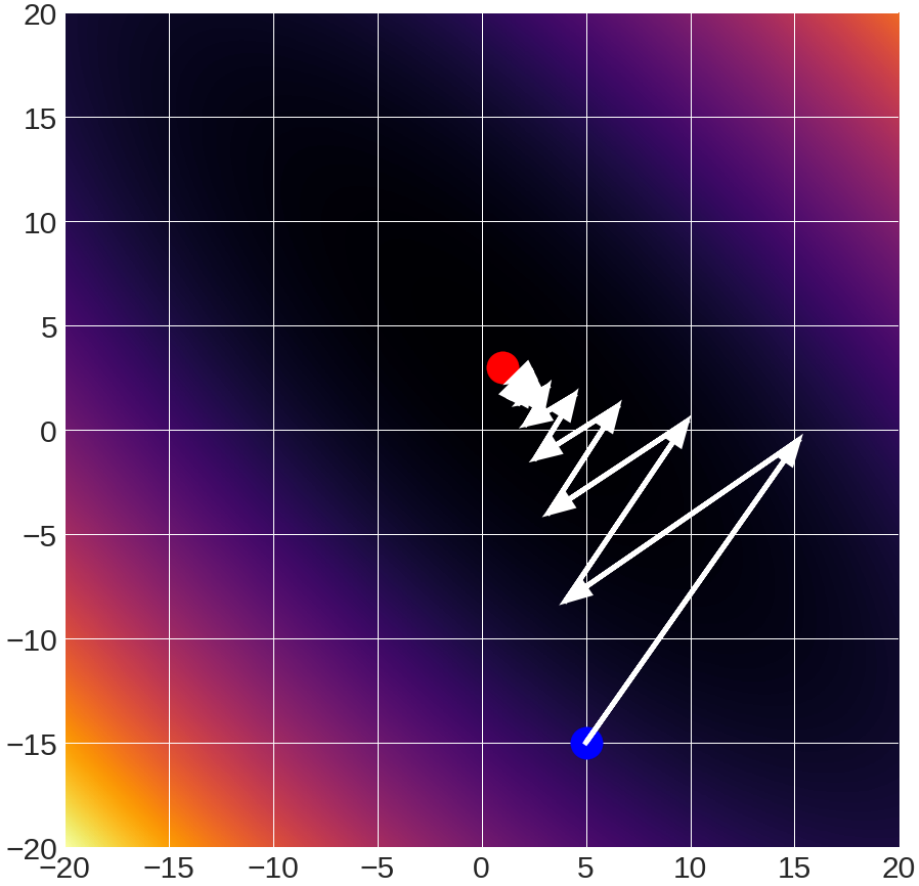
Method: at each step, move in direction of negative gradient

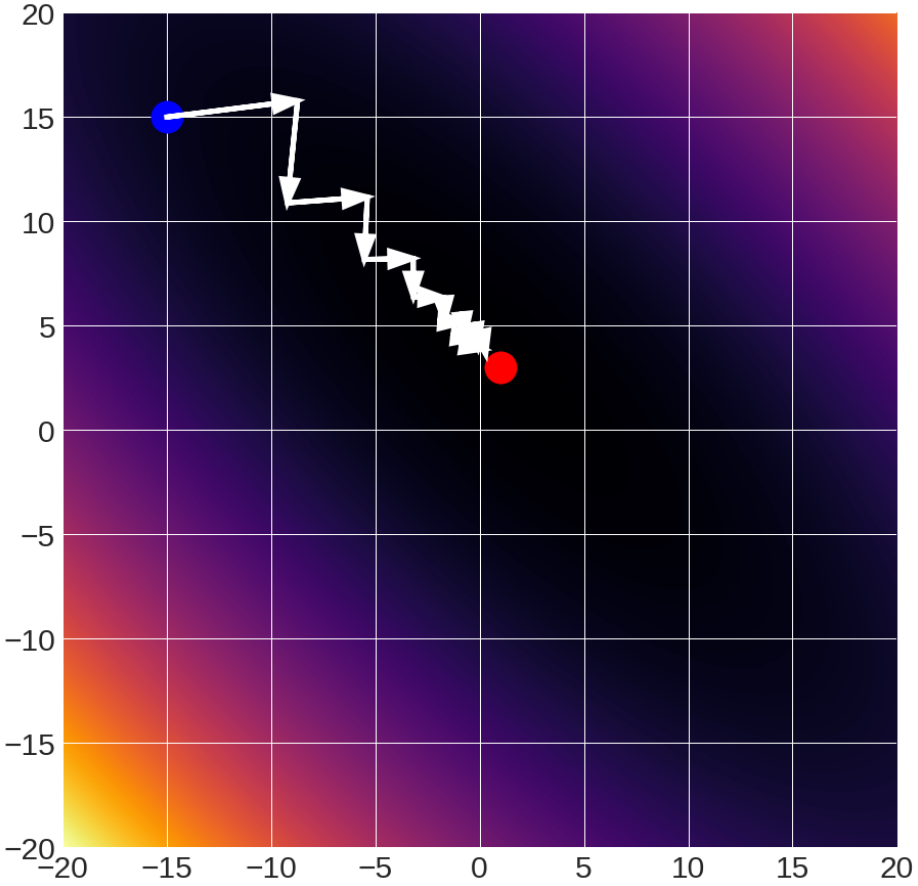
w0 = *initialize***()** #initialize
for iter in range(numIters):
    g = $\nabla_{\boldsymbol{w}} L(\boldsymbol{w})$ #eval gradient
    w = **w** + -*stepsize*(iter)***g** #update w
return **w**

Gradient Descent

Given starting point (blue)

$w_{i+1} = w_i + -9.8x10^{-2} \times \text{gradient}$





Computing the Gradient

How Do You Compute The Gradient?

Numerical Method:

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial L(w)}{\partial x_1} \\ \vdots \\ \frac{\partial L(w)}{\partial x_n} \end{bmatrix}$$

How do you compute this?

$$\frac{\partial f(x)}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

In practice, use:

$$\frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

Computing the Gradient

How Do You Compute The Gradient?

Numerical Method:

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial L(w)}{\partial x_1} \\ \vdots \\ \frac{\partial L(w)}{\partial x_n} \end{bmatrix}$$

Use: $\frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$

How many function evaluations per dimension?

Computing the Gradient

How Do You Compute The Gradient?
Analytical Method:

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial L(w)}{\partial x_1} \\ \vdots \\ \frac{\partial L(w)}{\partial x_n} \end{bmatrix}$$

Use calculus!

Computing the Gradient

$$L(\boldsymbol{w}) = \lambda \|\boldsymbol{w}\|_2^2 + \sum_{i=1}^{n} \left(y_i - \boldsymbol{w}^T \boldsymbol{x_i}\right)^2$$

$$\frac{\partial}{\partial \boldsymbol{w}}$$

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = 2\lambda \boldsymbol{w} + \sum_{i=1}^{n} -(2(y_i - \boldsymbol{w}^T \boldsymbol{x_i})\boldsymbol{x}_i)$$

Note: if you look at other derivations, things are written either $(y\text{-}w^Tx)$ or $(w^Tx - y)$; the gradients will differ by a minus.

Interpreting the Gradient

Recall:

w = **w** + -$\nabla_{\boldsymbol{w}} L(\boldsymbol{w})$ #update w

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = 2\lambda \boldsymbol{w} + \sum_{i=1}^{n} -(2(y_i - \boldsymbol{w}^T \boldsymbol{x_i})\boldsymbol{x_i})$$

Push **w** towards 0

$$-\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = -2\lambda \boldsymbol{w} + \sum_{i=1}^{n} (2(y_i - \boldsymbol{w}^T \boldsymbol{x_i})\boldsymbol{x_i})$$

α

If $y_i > w^T x_i$ (too *low*): then $w = w + \alpha x_i$ for some α

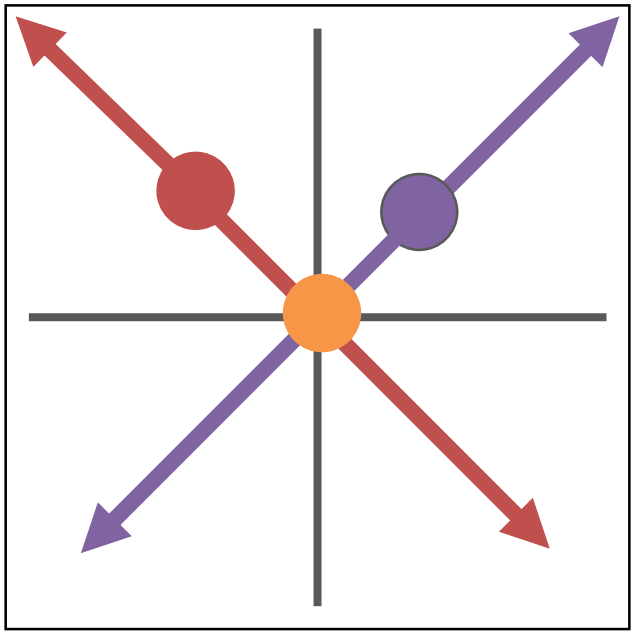
Before: $w^T x$

After: $(w + \alpha x)^T x = w^T x + \alpha x^T x$

Quick annoying detail: subgradients

What is the derivative of |x|?

|x|



Derivatives/Gradients

Defined everywhere but 0

$$\frac{\partial}{\partial x} f(x) = \text{sign}(x) \quad x \neq 0$$

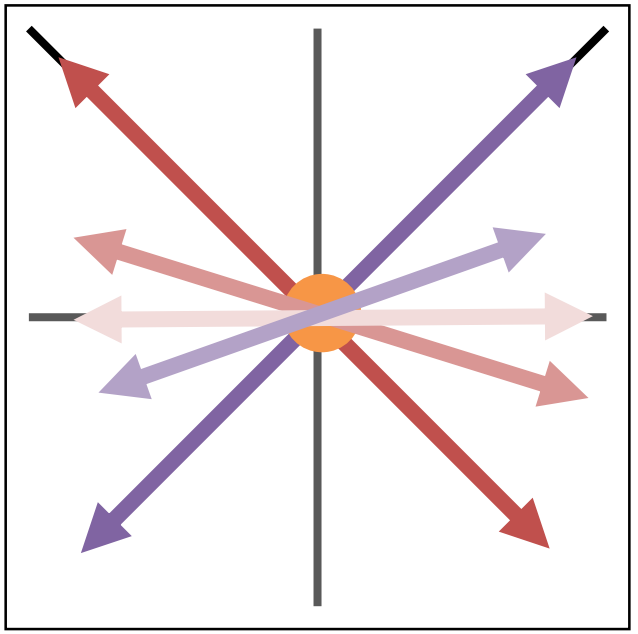
undefined $\quad x = 0$

Oh no! A discontinuity!

Quick annoying detail: subgradients

Subgradient: any underestimate of function

|x|



Subderivatives/subgradients

Defined everywhere

$$\frac{\partial}{\partial x} f(x) = \text{sign}(x) \quad x \neq 0$$

$$\frac{\partial}{\partial x} f(x) \in [-1,1] \quad x = 0$$

In practice: at discontinuity, pick value on either side.

Let’s Compute Another Gradient

Computing The Gradient

Multiclass Support Vector Machine

$$\arg\min_{\boldsymbol{W}} \boldsymbol{\lambda}\|\boldsymbol{W}\|_2^2 + \sum_{i=1}^{n}\sum_{j\neq y_i} \max(0, (\boldsymbol{W}\boldsymbol{x}_i)_j - (\boldsymbol{W}\boldsymbol{x}_i)_{y_i} + m)$$

Notation:

W $\rightarrow$ rows w_i (i.e., per-class scorer)

$(Wx_i)_j \rightarrow w_j^T x_i$

$$\arg\min_{\boldsymbol{W}} \boldsymbol{\lambda}\sum_{j=1}^{K}\|\boldsymbol{w}_j\|_2^2 + \sum_{i=1}^{n}\sum_{j\neq y_i} \max(0, \boldsymbol{w}_j^T\boldsymbol{x}_i - \boldsymbol{w}_{y_i}^T\boldsymbol{x}_i + m)$$

Derivation setup: Karpathy and Fei-Fei

Computing The Gradient

$$\arg\min_{\boldsymbol{W}} \boldsymbol{\lambda} \sum_{\boldsymbol{j=1}}^{\boldsymbol{K}} \|\boldsymbol{w_j}\|_{\boldsymbol{2}}^{\boldsymbol{2}} + \sum_{i=1}^{n} \underbrace{\sum_{j \neq y_i} \max(0, \boldsymbol{w}_j^T \boldsymbol{x}_i - \boldsymbol{w}_{y_i}^T \boldsymbol{x}_i + m)}$$

$$\frac{\partial}{\partial w_j}:$$

$$w_j^T x_i - w_{y_i}^T x_i + m \leq 0: \quad 0$$

$$w_j^T x_i - w_{y_i}^T x_i + m > 0: \quad x_i$$

$$\rightarrow 1(w_j^T \ x_i - w_{y_i}^T x_i + m > 0) x_i$$

Derivation setup: Karpathy and Fei-Fei

Computing The Gradient

$$\arg\min_{\boldsymbol{W}} \boldsymbol{\lambda} \sum_{\boldsymbol{j=1}}^{\boldsymbol{K}} \|\boldsymbol{w_j}\|_{\boldsymbol{2}}^{\boldsymbol{2}} + \sum_{i=1}^{n} \underbrace{\sum_{j \neq y_i} \max(0, \boldsymbol{w}_j^T \boldsymbol{x}_i - \boldsymbol{w}_{y_i}^T \boldsymbol{x}_i + m)}$$

$$\frac{\partial}{\partial w_{y_i}}: \quad \sum_{j \neq y_i} 1\left(w_j^T x_i - w_{y_i}^T x_i + m > 0\right)(-x_i)$$

Derivation setup: Karpathy and Fei-Fei

Interpreting The Gradient

$$-\frac{\partial}{\partial w_j}: \underbrace{-1(w_j^T x_i - w_{y_i}^T x_i + m > 0)}_{\text{If we do not predict the correct class by at least a score difference of m …}} \underbrace{x_i}_{\text{Want incorrect class's scoring vector to score that point lower.}}$$

Recall:

Before: $w^T x$;

After: $(w-\alpha x)^T x = w^T x - a x^T x$

Derivation setup: Karpathy and Fei-Fei

Computing The Gradient

- Numerical: foolproof but slow
- Analytical: can mess things up ☺
- In practice: do analytical, but check with numerical (called a gradient check)

Slide: Karpathy and Fei-Fei

Implementing Gradient Descent

Loss is a function that we can evaluate over data

All Data

$$-\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = -2\lambda \boldsymbol{w} + \sum_{i=1}^{n} (2(y_i - \boldsymbol{w}^T \boldsymbol{x_i}) \boldsymbol{x}_i)$$

Subset B

$$-\nabla_{\boldsymbol{w}} L_B(\boldsymbol{w}) = -2\lambda \boldsymbol{w} + \sum_{i \in B} (2(y_i - \boldsymbol{w}^T \boldsymbol{x_i}) \boldsymbol{x}_i)$$

Implementing Gradient Descent

Option 1: Vanilla Gradient Descent
Compute gradient of L over all data points

```
for iter in range(numIters):
    g = gradient(data,L)
    w = w + -stepsize(iter)*g #update w
```

Implementing Gradient Descent

Option 2: *Stochastic* Gradient Descent
Compute gradient of L over 1 random sample

```
for iter in range(numIters):
    index = randint(0,#data)
    g = gradient(data[index],L)
    w = w + -stepsize(iter)*g #update w
```

Implementing Gradient Descent

Option 3: *Minibatch* Gradient Descent

Compute gradient of L over subset of B samples

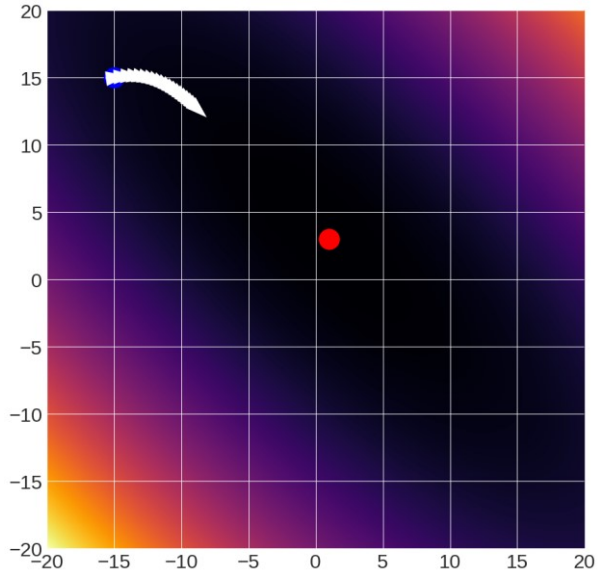
```
for iter in range(numIters):
    subset = choose_samples(#data,B)
    g = gradient(data[subset],L)
    w = w + -stepsize(iter)*g #update w
```

Typical batch sizes: ~100

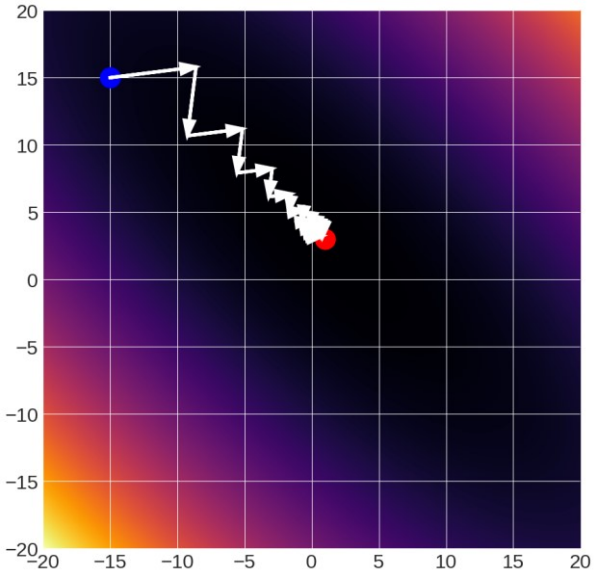
Gradient Descent Details

Step size (also called **learning rate** / **lr**)
critical parameter

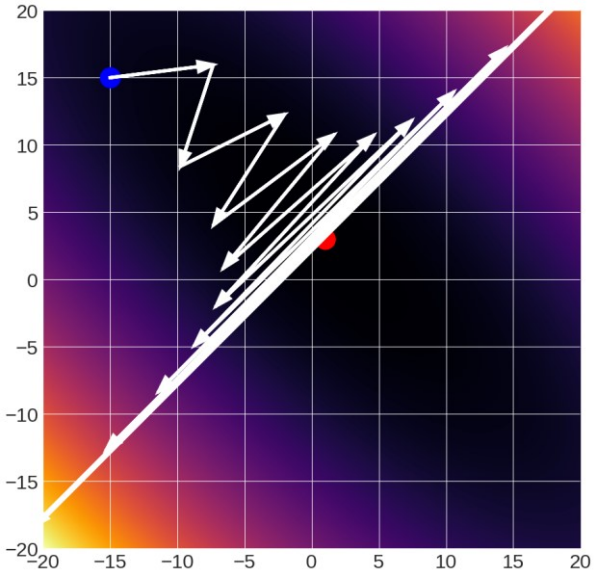
$1x10^{-2}$
falls short



$10x10^{-2}$
converges

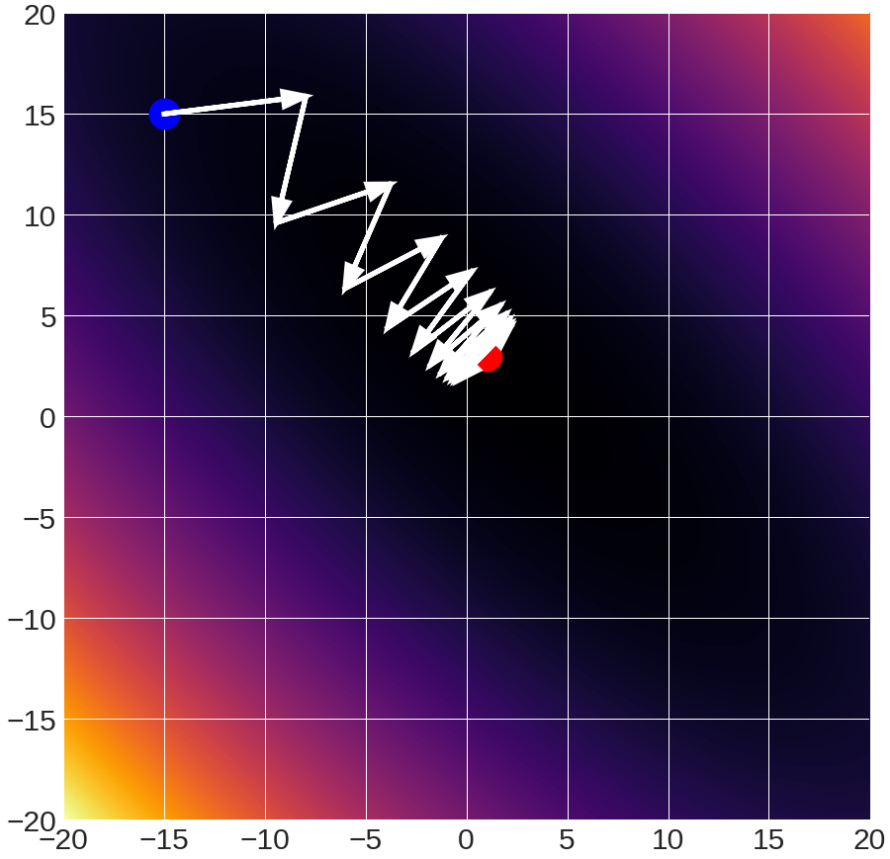


$12x10^{-2}$
diverges



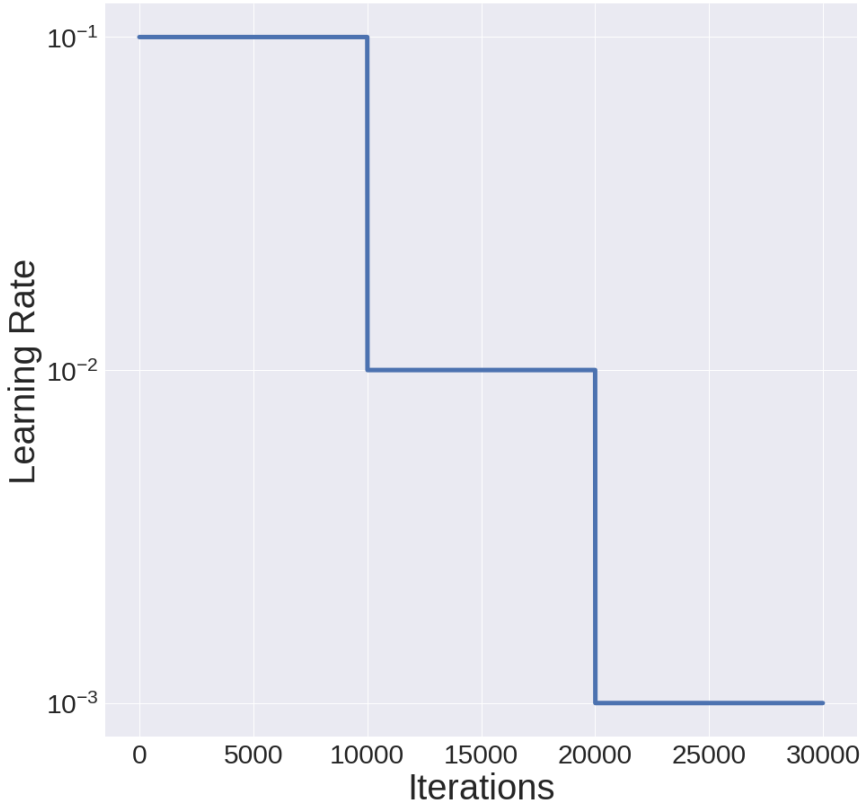
Gradient Descent Details

$11x10^{-2}$:oscillates
(Raw gradients)



Gradient Descent Details

Typical solution: start with initial rate lr, multiply by f every N interations



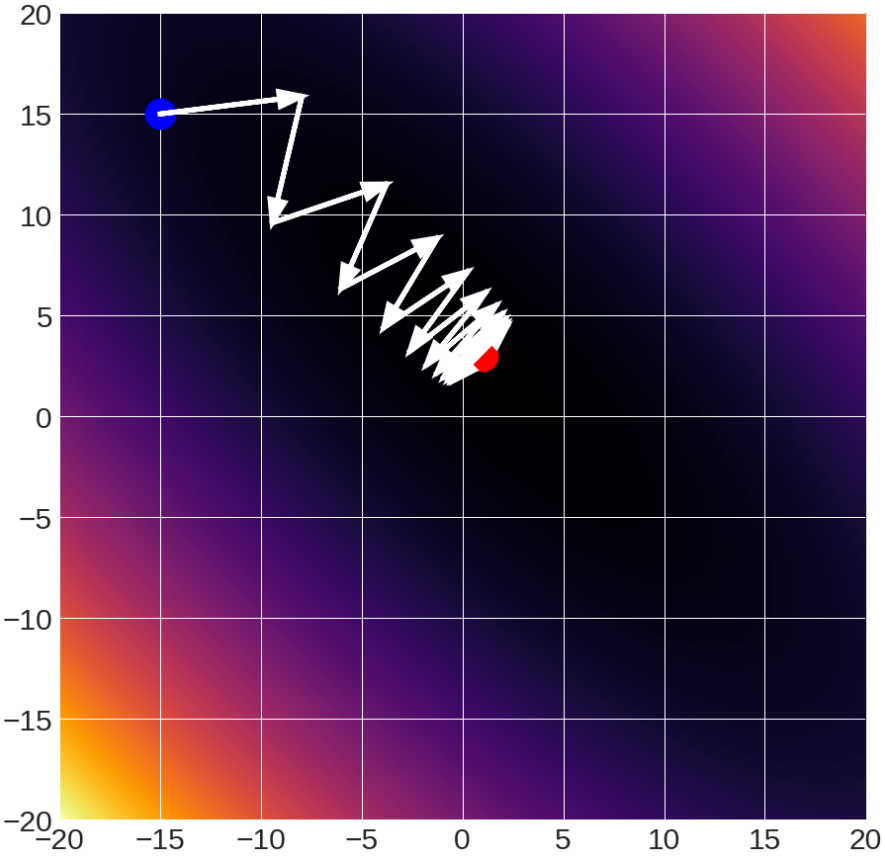
init_lr = 10^{-1}

f = 0.1

N = 10K

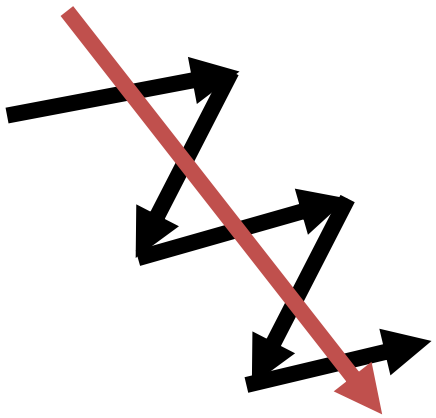
Gradient Descent Details

11×10^{-2} :oscillates
(Raw gradients)



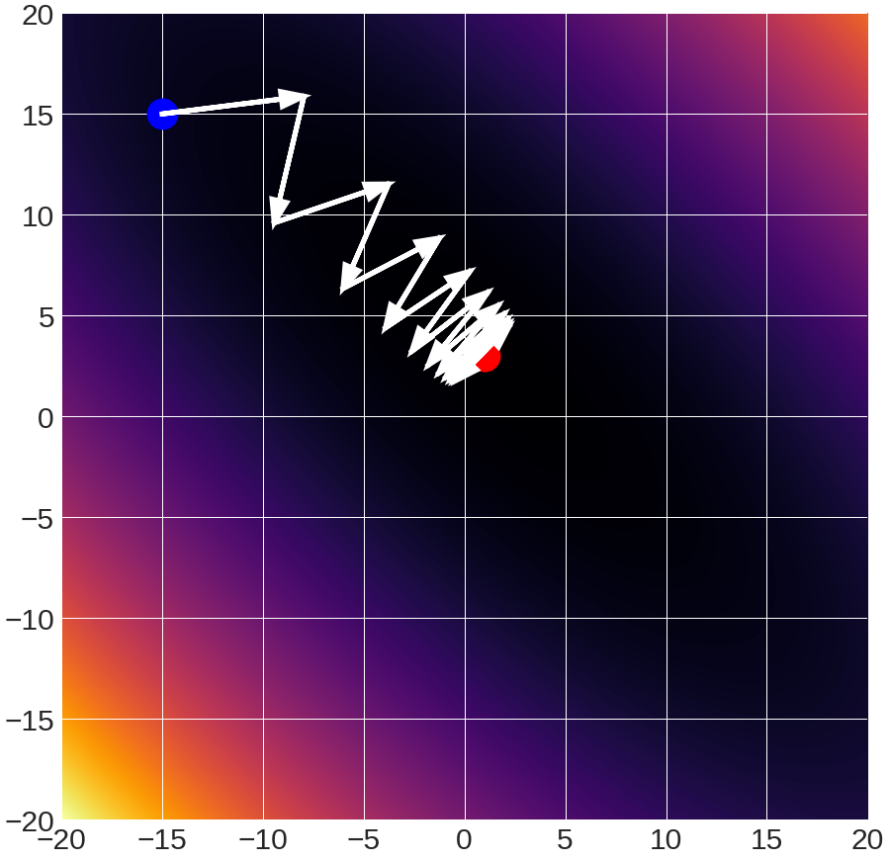
Solution:
Average gradients

With exponentially decaying weights, called “momentum”

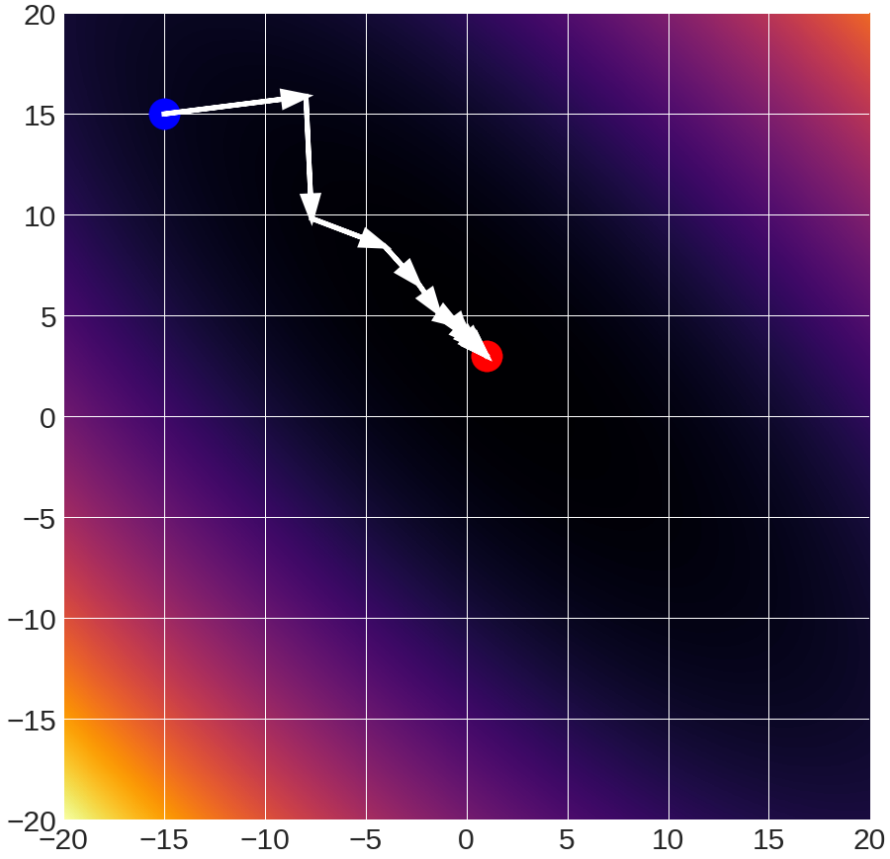


Gradient Descent Details

$11x10^{-2}$:oscillates
(Raw gradients)



$11x10^{-2}$
(0.25 momentum)

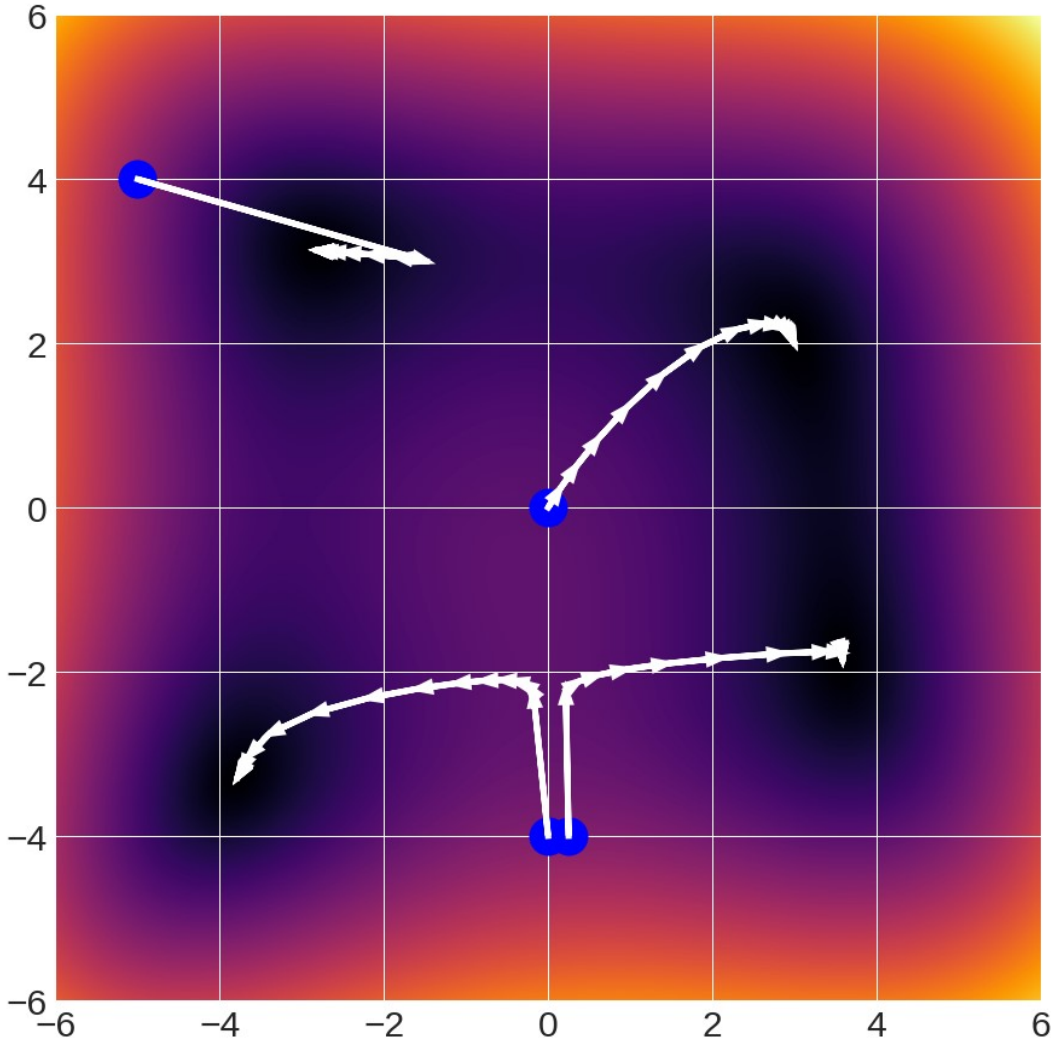


Gradient Descent Details

Multiple Minima

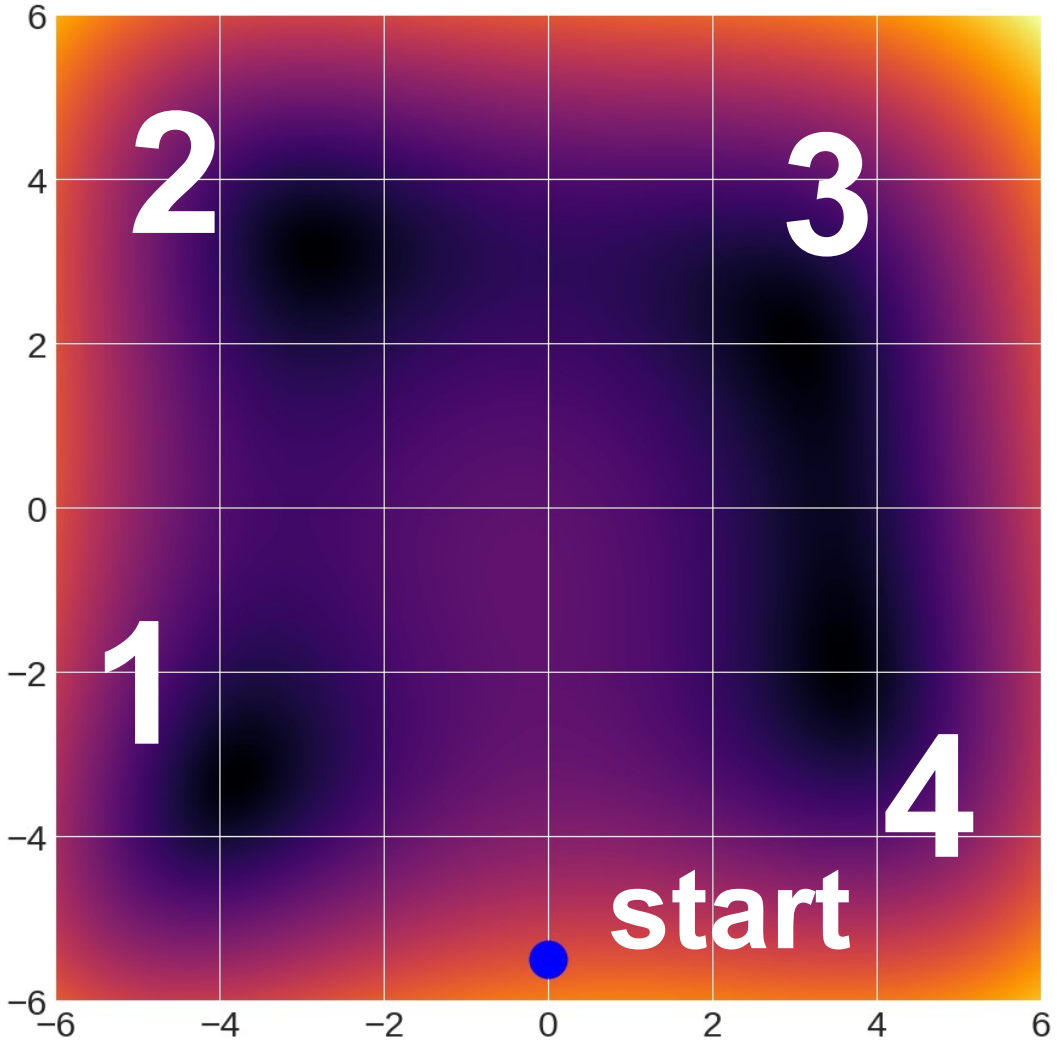
→

Gradient Descent Finds **local minimum**

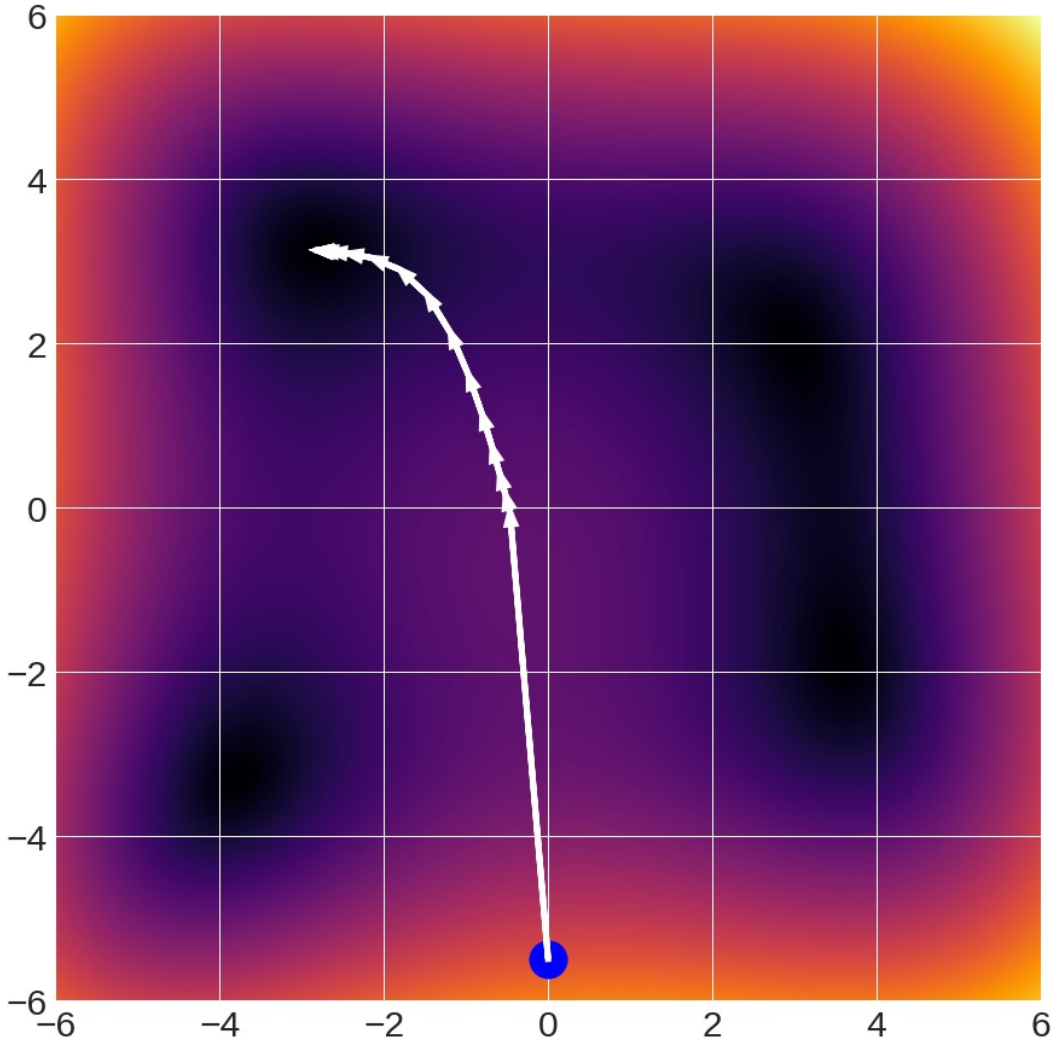


Gradient Descent Details

Guess the minimum!

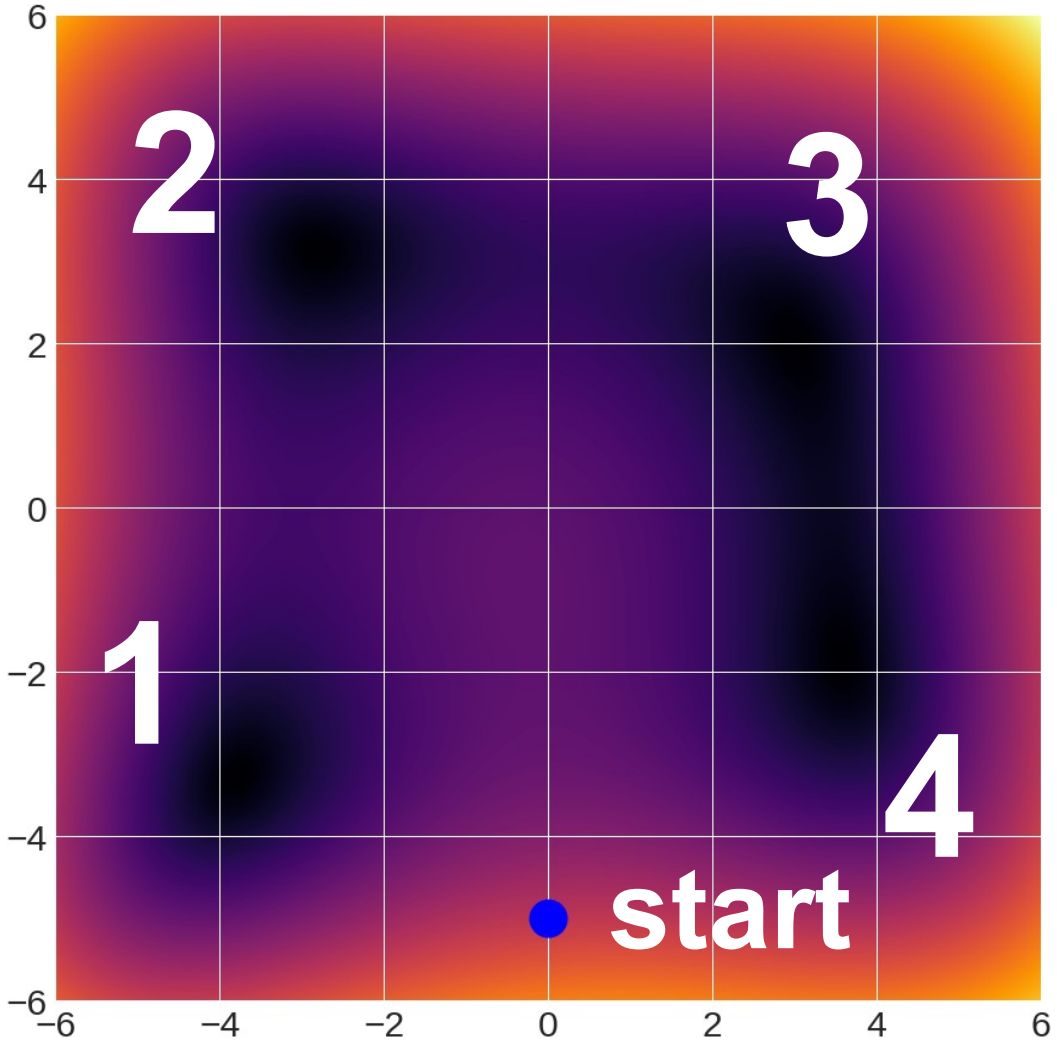


Gradient Descent Details



Gradient Descent Details

Guess the minimum!

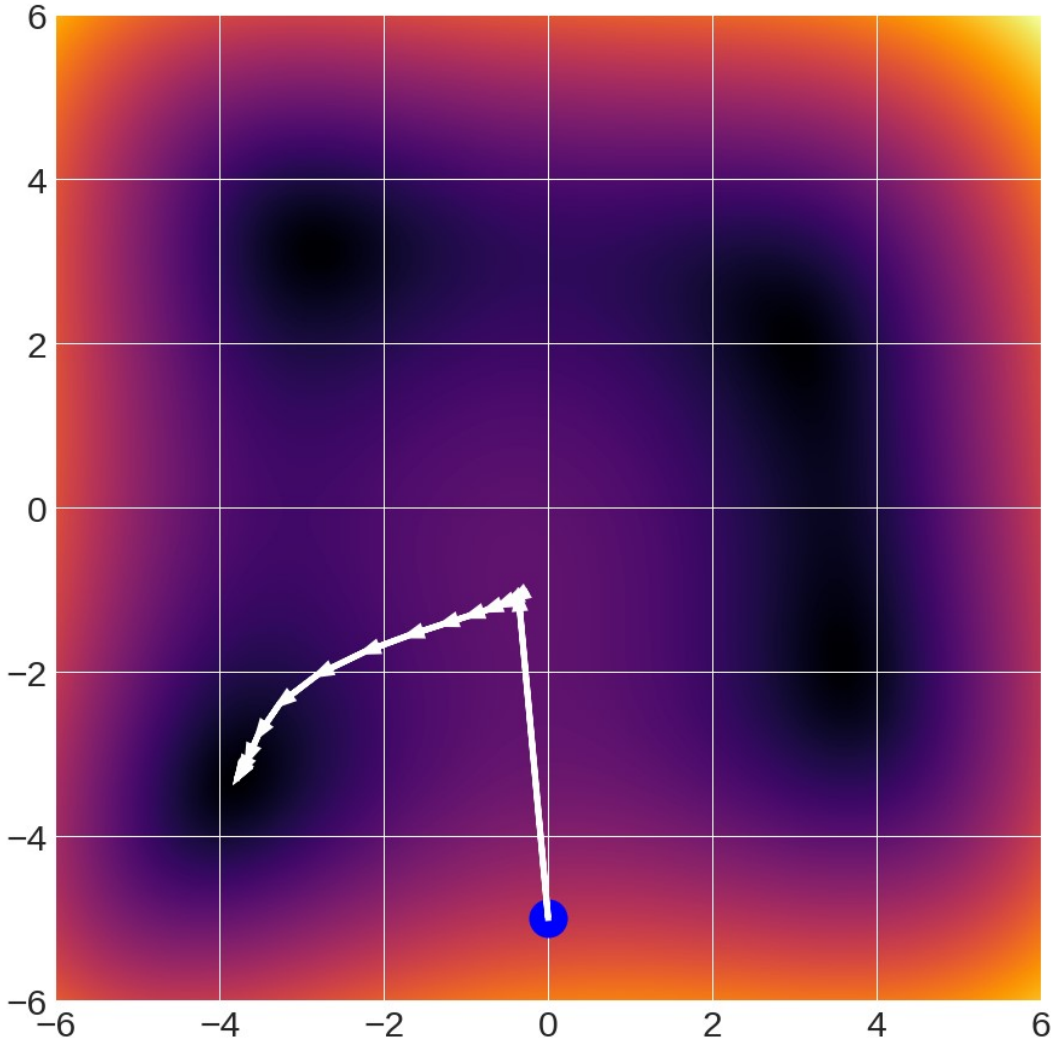


Gradient Descent Details

Dynamics are fairly complex

Many important functions are **convex**: any local minimum is a global minimum

Many important functions are not.



In practice

- **Conventional wisdom**: minibatch stochastic gradient descent (SGD) + momentum (package implements it for you) + decaying learning rate
- The above is typically what is meant by “SGD”
- Other update rules exist; benefits in general not clear (sometimes better, sometimes worse)

Optimizing Everything

$$L(\boldsymbol{W}) = \boldsymbol{\lambda}\|\boldsymbol{W}\|_{\boldsymbol{2}}^{\boldsymbol{2}} + \sum_{i=1}^{n} -\log\left(\frac{\exp((Wx)_{y_i})}{\sum_k \exp((Wx)_k))}\right)$$

$$L(\boldsymbol{w}) = \lambda\|\boldsymbol{w}\|_2^2 + \sum_{i=1}^{n}\left(y_i - \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}}\right)^2$$

- Optimize **w** on training set with SGD to maximize training accuracy
- Optimize λ with random/grid search to maximize validation accuracy
- Note: Optimizing λ on training sets it to 0

(Over/Under)fitting and Complexity

Let's fit a polynomial: given x, predict y

Note: can do non-linear regression with copies of x

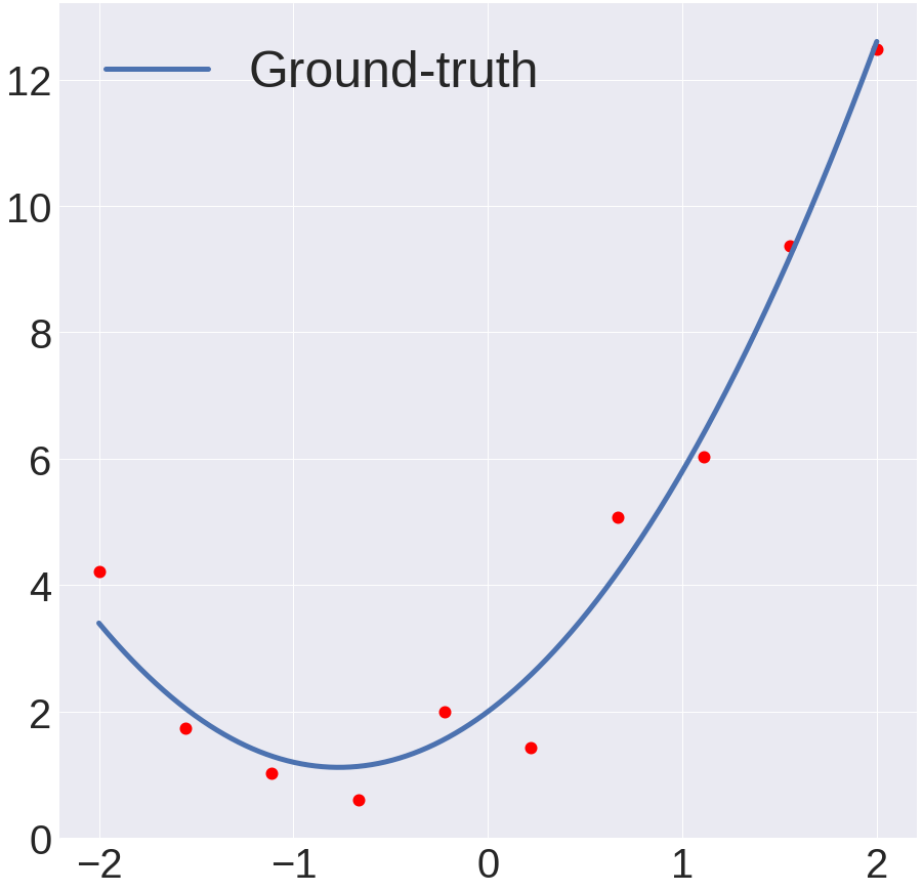
$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} x_1^F & \cdots & x_1^2 & x_1 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ x_N^F & \cdots & x_N^2 & x_N & 1 \end{bmatrix} \begin{bmatrix} w_F \\ \vdots \\ w_2 \\ w_1 \\ w_0 \end{bmatrix}$$

Matrix of all polynomial degrees

Weights: one per polynomial degree

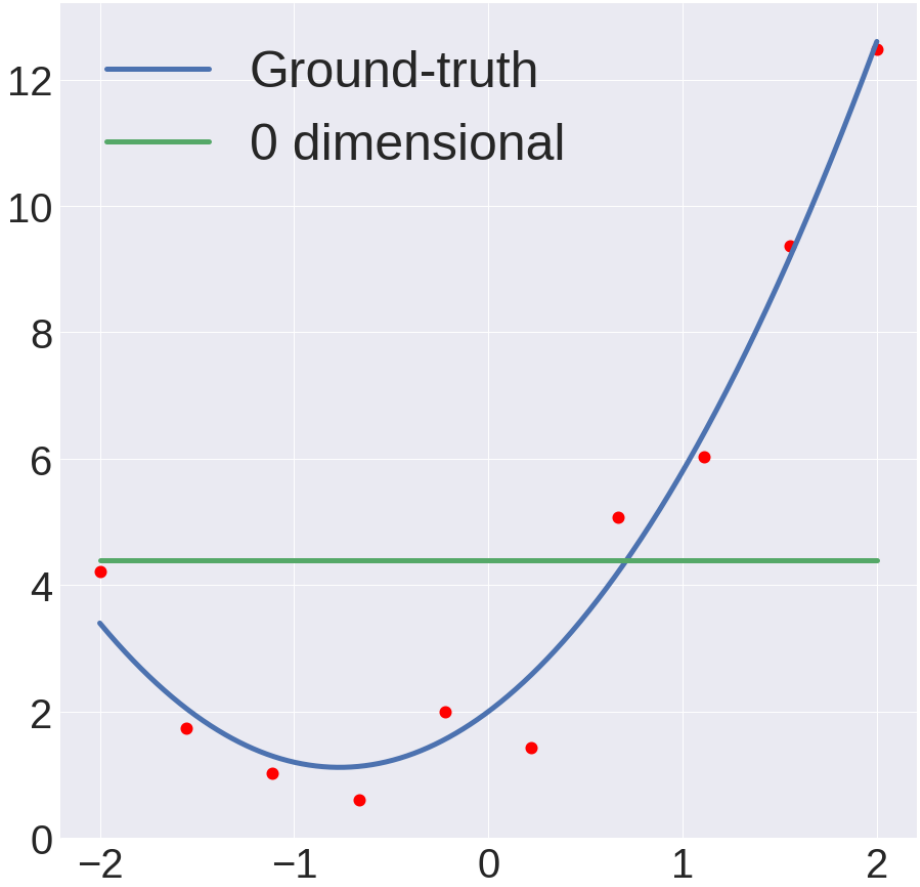
(Over/Under)fitting and Complexity

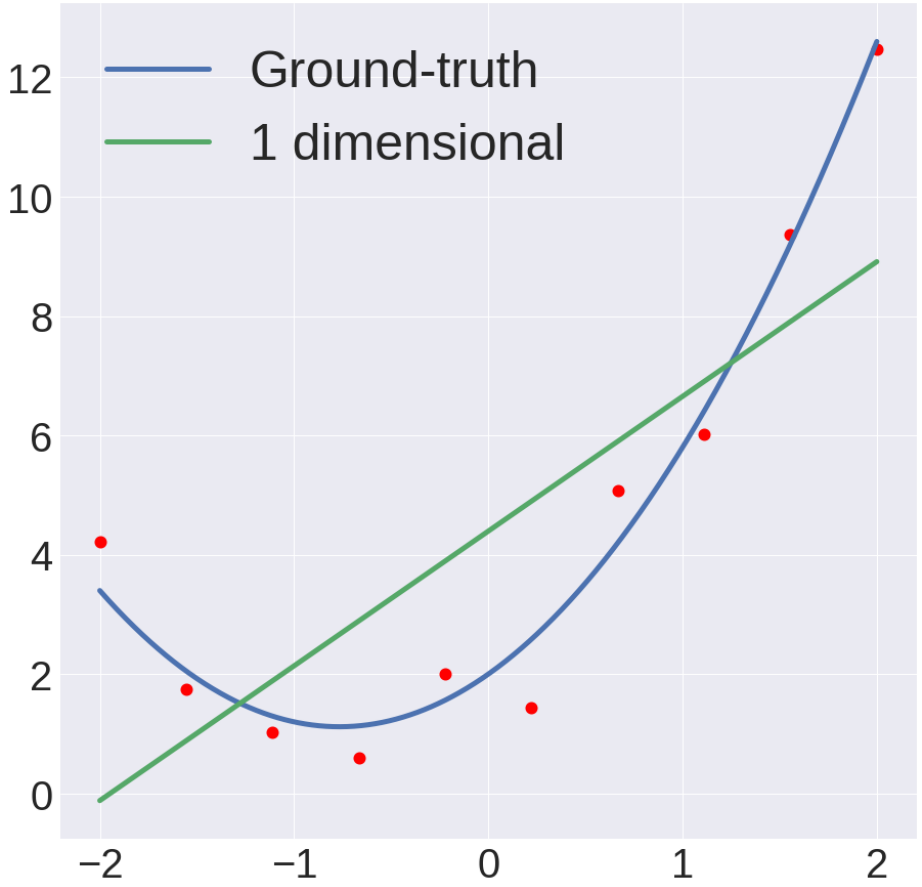
Model: $1.5x^2 + 2.3x+2 + N(0,0.5)$



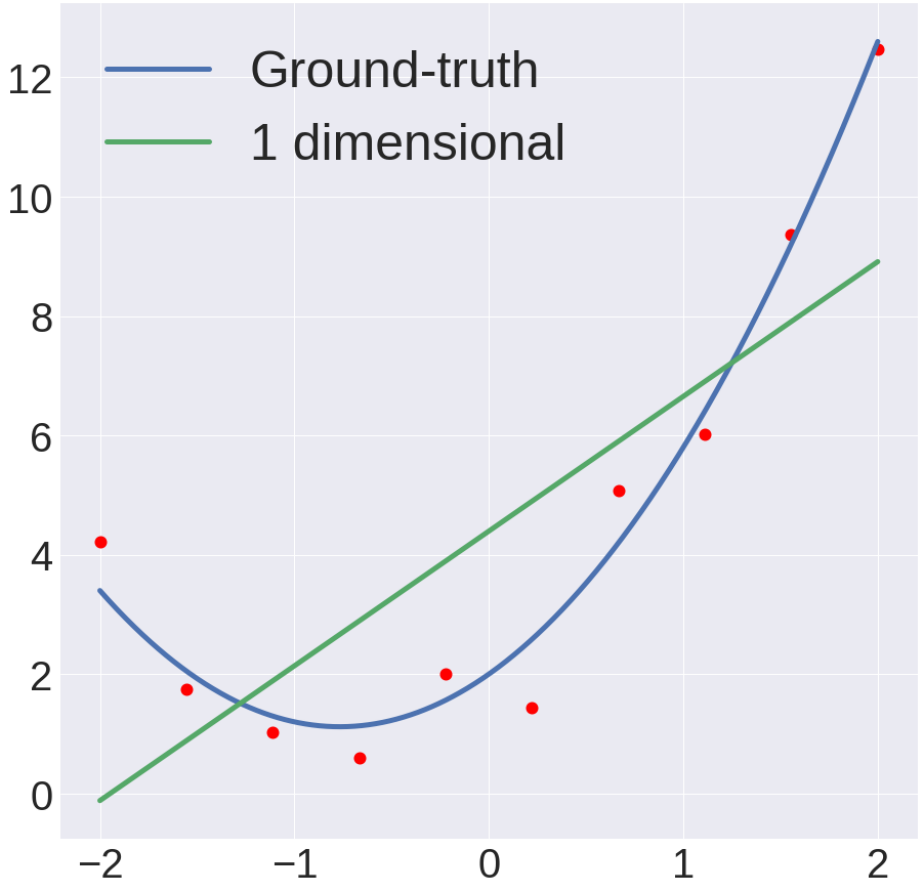
Underfitting

Model: $1.5x^2 + 2.3x+2 + N(0,0.5)$





Underfitting

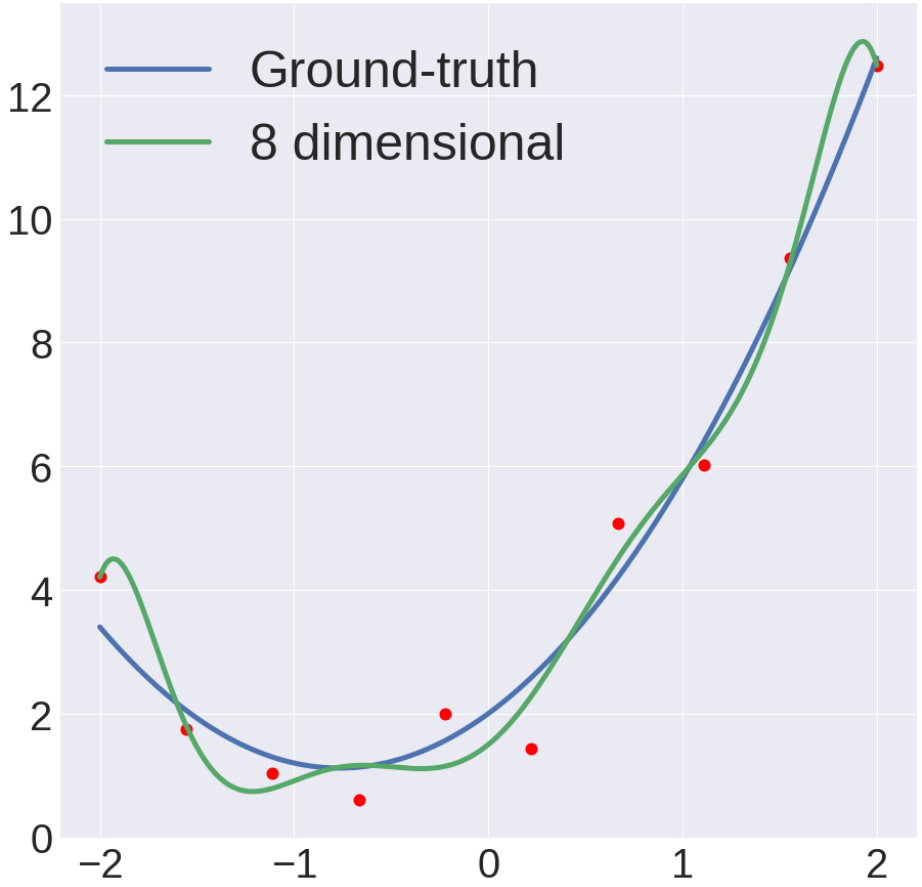


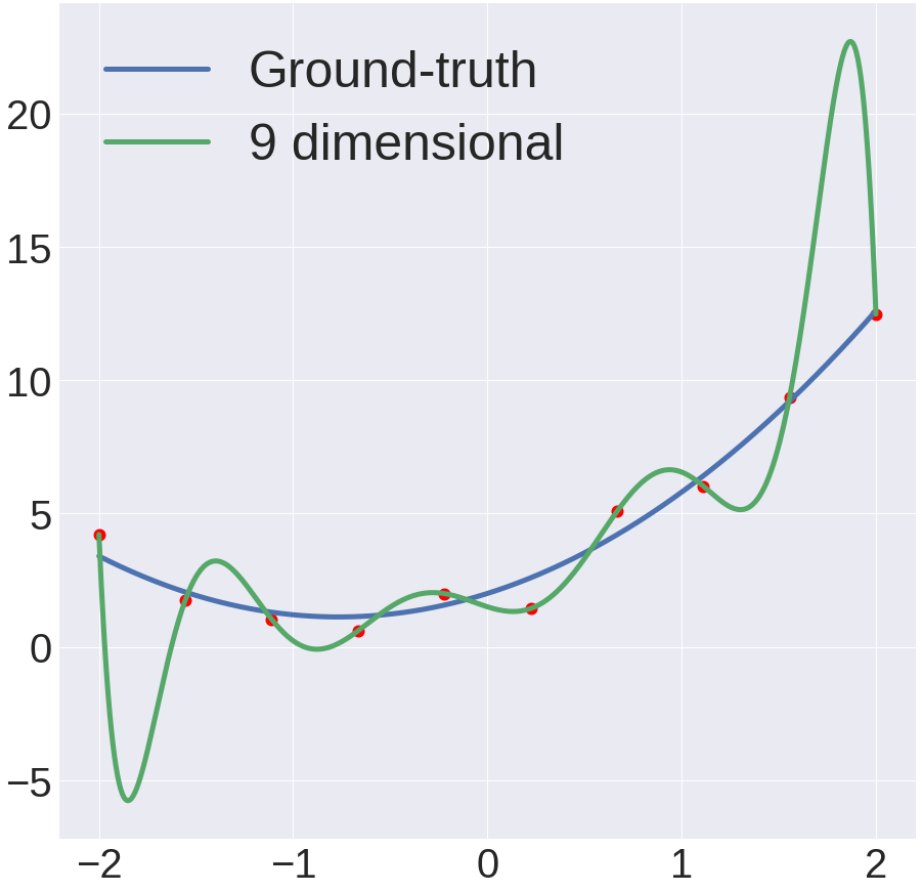
Model doesn’t have the parameters to fit the data.

Bias (statistics): Error intrinsic to the model.

Overfitting

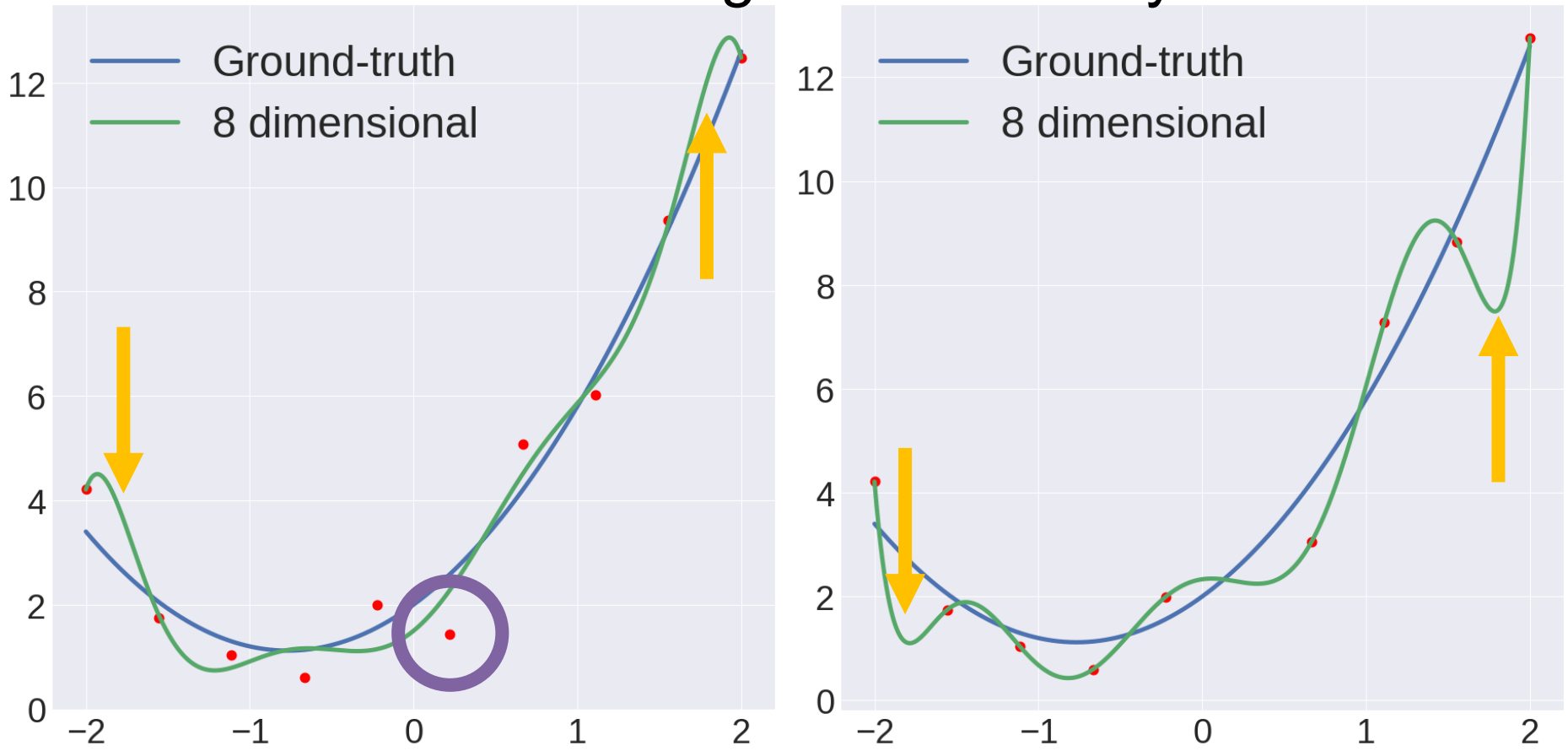
Model: $1.5x^2 + 2.3x+2 + N(0,0.5)$





Overfitting

Model has high ***variance***: remove **one point**, and model changes dramatically



(Continuous) Model Complexity

$$\arg\min_{W} \lambda \|W\|_2^2 + \sum_{i=1}^{n} -\log\left(\frac{\exp((Wx)_{y_i})}{\sum_k \exp((Wx)_k)}\right)$$

Regularization: penalty for complex model

Pay penalty for negative log-likelihood of correct class

Intuitively: big weights = more complex model

Model 1: $0.01*x_1 + 1.3*x_2 + -0.02*x_3 + -2.1x_4 + 10$

Model 2: $37.2*x_1 + 13.4*x_2 + 5.6*x_3 + -6.1x_4 + 30$

Fitting Model

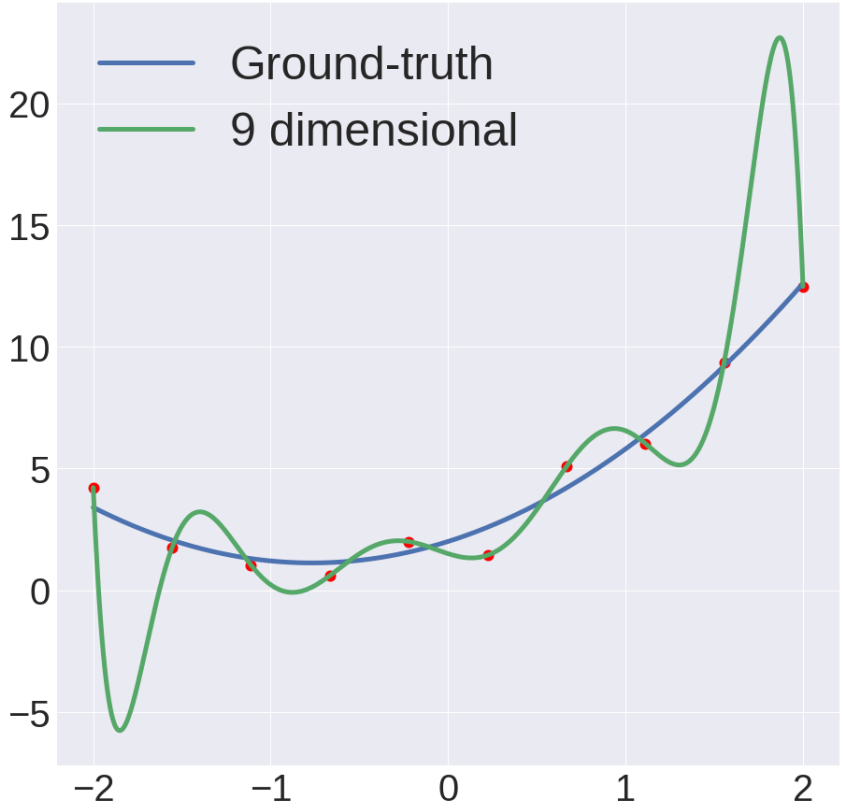
Again, fitting polynomial, but with regularization

$$\arg\min_{\mathbf{w}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\| + \lambda\|\boldsymbol{w}\|$$

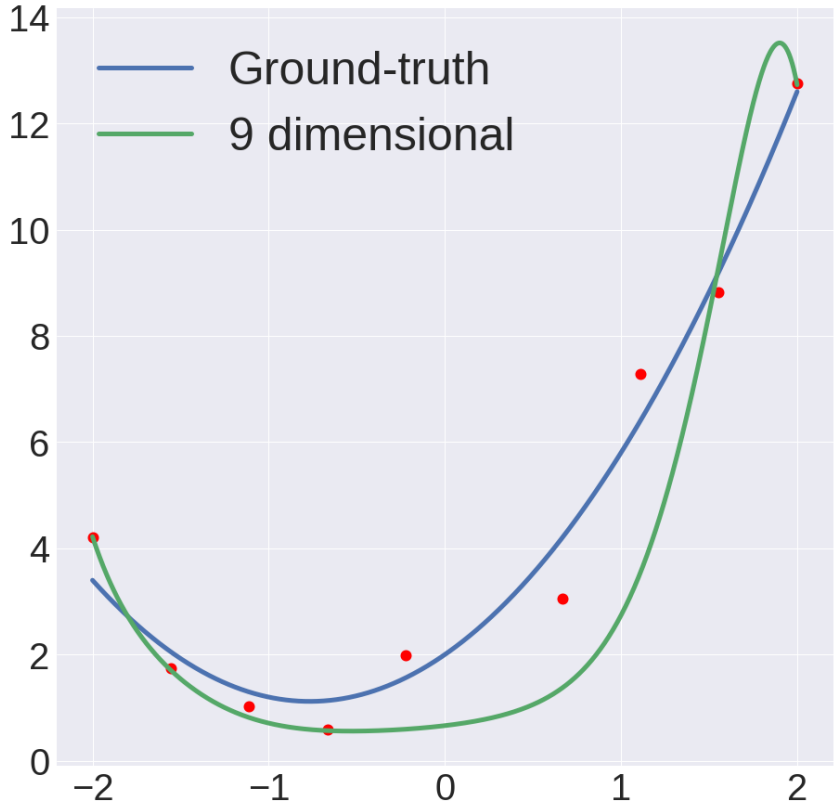
$$\begin{bmatrix} x_1^F & \cdots & x_1^2 & x_1 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ x_N^F & \cdots & x_N^2 & x_N & 1 \end{bmatrix} \quad \begin{bmatrix} w_F \\ \vdots \\ w_0 \end{bmatrix}$$

Adding Regularization

No regularization:
fits all data points



Regularization:
can’t fit all data points



In General

Error on new data comes from combination of:

1. Bias: model is oversimplified and can't fit the underlying data
2. Variance: you don't have the ability to estimate your model from limited data
3. Inherent: the data is intrinsically difficult

Bias and variance trade-off. Fixing one hurts the other.

Underfitting and Overfitting

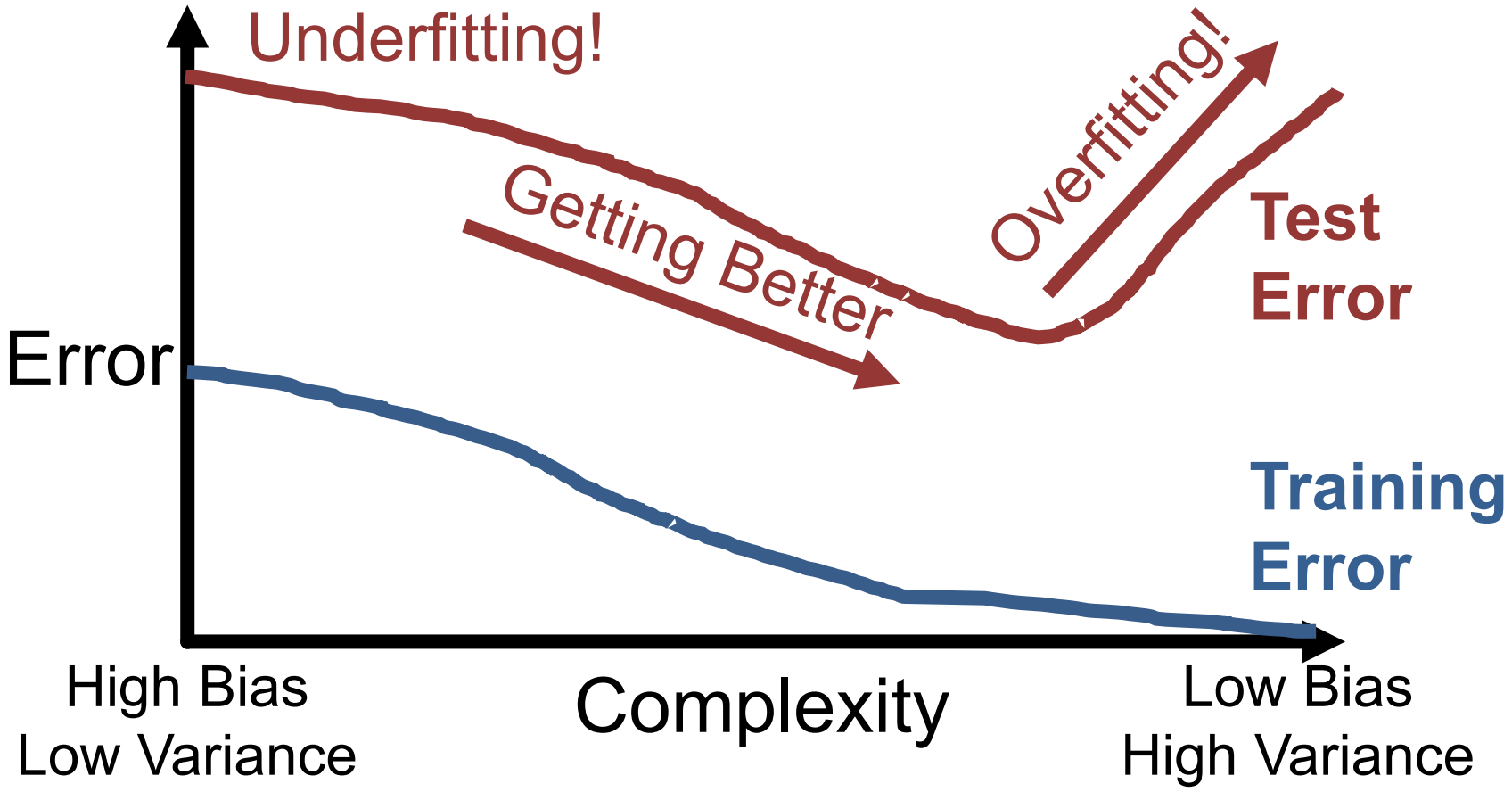


Diagram adapted from: D. Hoiem

Underfitting and Overfitting

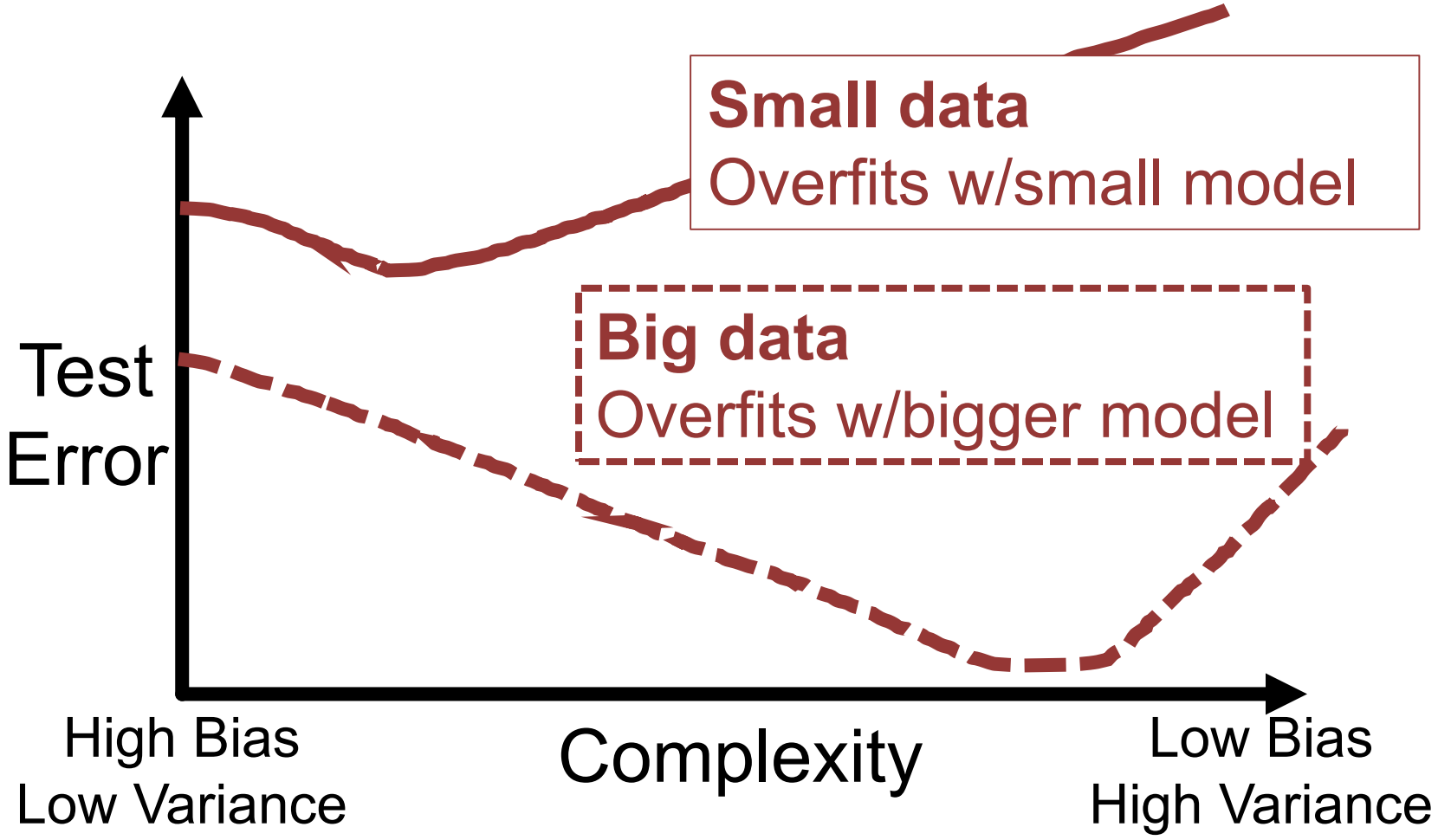
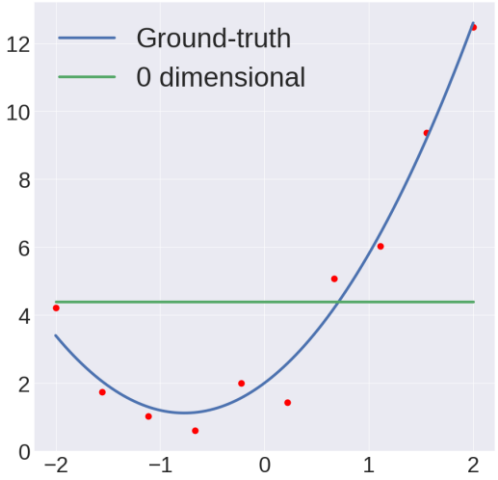
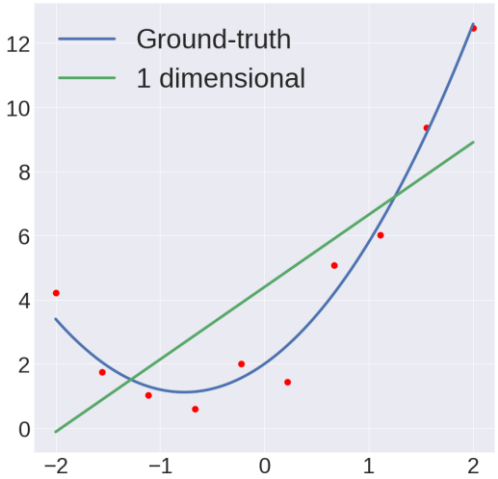


Diagram adapted from: D. Hoiem

Underfitting



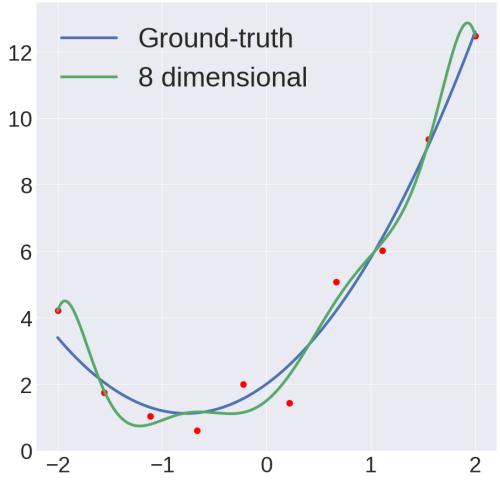


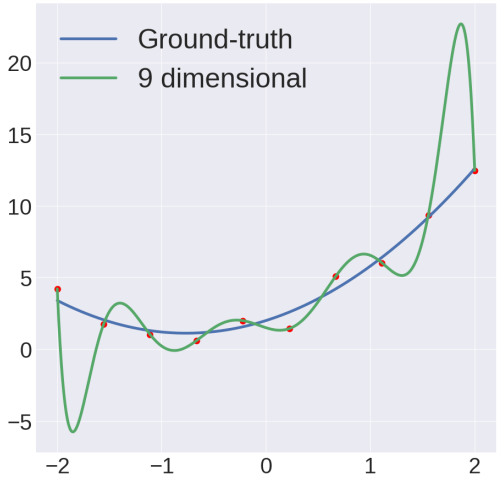
Do poorly on both training and validation data due to bias.

Solution:

1. More features
2. More powerful model
3. Reduce regularization

Overfitting





Do well on training data, but poorly on validation data due to variance

Solution:

1. More data
2. Less powerful model
3. Regularize your model more

Cris Dima rule: first make sure you *can* overfit, then stop overfitting.

Next Class

- Non-linear models (neural nets)