Lecture 12: Optimization

Administrative

- HW1 Grades Released
  - Submit regrade requests **via Gradescope** by **Friday 3/5**
  - Minor regrades (<1 point per question, <3 points overall) will be processed at the end of the semester only if they affect your final grade. Submit on Gradescope, **and** send an email to course staff with subject "EECS 442W21 Minor Regrade Request"

- HW1 Color Space & Illumination context
  - See entries here: https://web.eecs.umich.edu/~justincj/teaching/eecs442/resources/WI21-hw1-vote/
  - Vote here: https://forms.gle/vJrDzGVChbsLV6on6

- HW3 due **Wednesday 3/10**
  - One extra late day with HW3 release (up to 7 total)

Last Time: Image Classification

Input: image

Output: Assign image to one of a fixed set of categories

cat
bird
deer
dog
truck

Last Time: Nearest Neighbor

Known Images
Labels

Test
Image

$\boldsymbol{x}_1 \leftarrow \boxed{D(\boldsymbol{x}_1, \boldsymbol{x}_T)} \rightarrow \boldsymbol{x}_T$

...

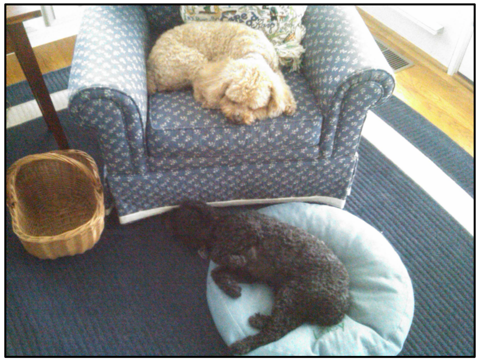
$\boxed{D(\boldsymbol{x}_N, \boldsymbol{x}_T)}$

$\boldsymbol{x}_N$

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

Last Time: Linear Classifiers

Example Setup: 3 classes

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

Want:

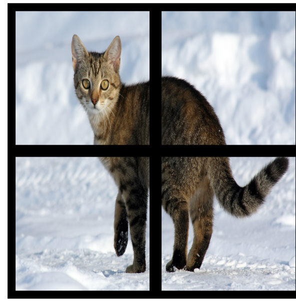
$\boldsymbol{w}_0^T\boldsymbol{x}$ big if cat

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

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

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

Last Time: Linear Classifiers



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

$\boldsymbol{W}$

56
231
24
2
1

$\boldsymbol{x_i}$

→

-96.8	Cat score
437.9	Dog score
61.95	Hippo score

$\boldsymbol{Wx_i}$

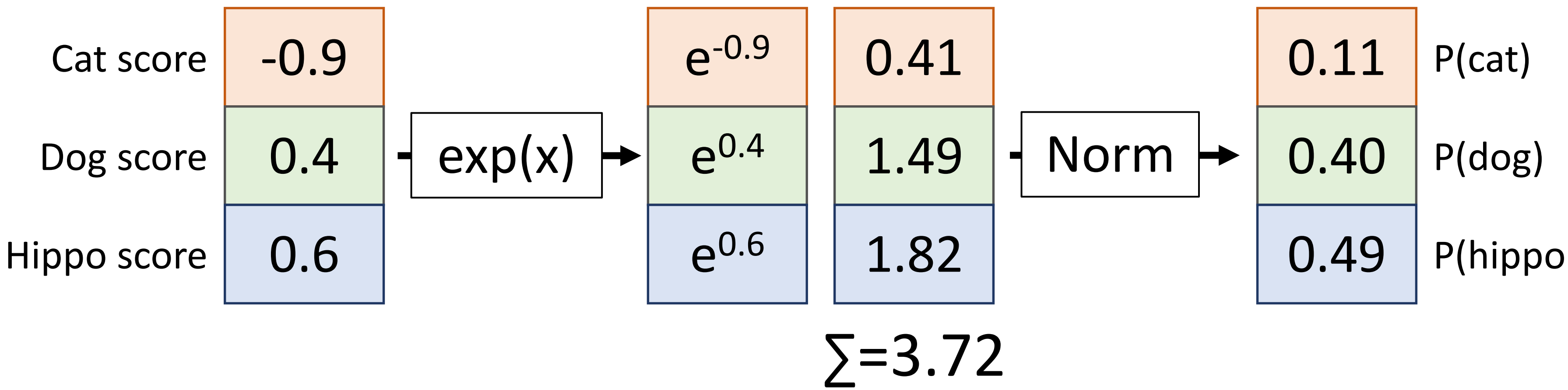
Weight matrix a collection of scoring functions, one per class

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

Diagram by: Karpathy, Fei-Fei

Last Time: Cross-Entropy Loss

Converting Scores to “Probability Distribution”



Generally P(class j):

$$\frac{\exp((Wx)_j)}{\sum_k \exp((Wx)_k)}$$

Called softmax function

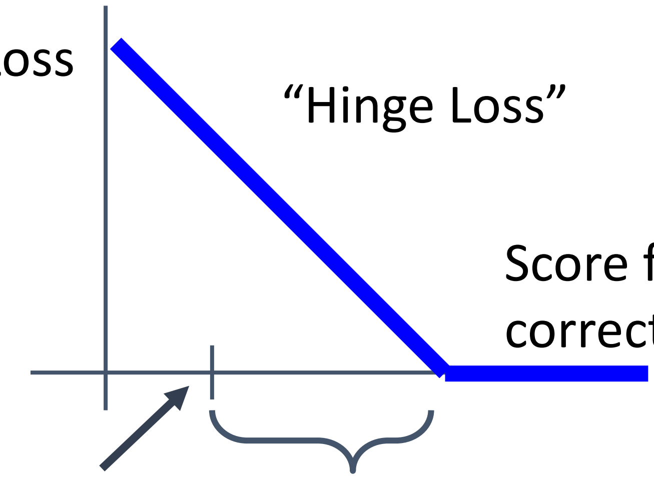
Loss is –log(P(correct class))

$$L_i = -\log \frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}$$

Today:
- Multiclass SVM loss
- Optimization

Multiclass SVM Loss

"The score of the correct class should be higher than all the other scores"



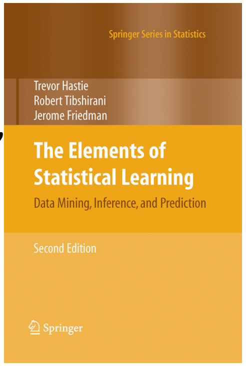
Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

SVM = "Support Vector Machine"
Lots of great theory about why this is a good idea – see EECS 445/545 or this book for more:



https://web.stanford.edu/~hastie/ElemStatLearn/

Multiclass SVM Loss



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Multiclass SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Multiclass SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9		

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

= max(0, 5.1 - 3.2 + 1)
+ max(0, -1.7 - 3.2 + 1)
= max(0, 2.9) + max(0, -3.9)
= 2.9 + 0
= 2.9

Multiclass SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

= max(0, 1.3 - 4.9 + 1)
+max(0, 2.0 - 4.9 + 1)
= max(0, -2.6) + max(0, -1.9)
= 0 + 0
= 0

Multiclass SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

= max(0, 2.2 - (-3.1) + 1)
+max(0, 2.5 - (-3.1) + 1)
= max(0, 6.3) + max(0, 6.6)
= 6.3 + 6.6
= 12.9

Multiclass SVM Loss



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Loss over the dataset is:

L = (2.9 + 0.0 + 12.9) / 3
= 5.27

Multiclass SVM Loss



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: What happens to the loss if the scores for the car image change a bit?

Multiclass SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: What are the min and max possible loss?

Multiclass SVM Loss

			
cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: If all scores were random, what loss would we expect?

Multiclass SVM Loss



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: What would happen if sum were over all classes? (including $j = y_i$)

Multiclass SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: What if the loss used mean instead of sum?

Multiclass SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Loss	2.9	0	12.9

Given an example (x_i, y_i)
(x_i is image, y_i is label)

Let $s = f(x_i, W)$ be scores

Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: What if we used this loss instead?

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$

Cross-Entropy vs SVM Loss

$$L_i = -\log \frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Assume scores:

[10, -2, 3]

[10, 9, 9]

[10, -100, -100]

and $y_i = 0$

Q: What is cross-entropy loss? What is SVM loss?

A: Cross-entropy loss > 0

SVM loss = 0

Cross-Entropy vs SVM Loss

$$L_i = -\log \frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Assume scores:
[10, -2, 3]
[10, 9, 9]
[10, -100, -100]
and $y_i = 0$

Q: What happens to each loss if I slightly change the scores of the last datapoint?

A: Cross-entropy loss will change; SVM loss will stay the same

Cross-Entropy vs SVM Loss

$$L_i = -\log \frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Assume scores:

[10, -2, 3]

[10, 9, 9]

[10, -100, -100]

and $y_i = 0$

Q: What happens to each loss if I double the score of the correct class from 10 to 20?

A: Cross-entropy loss will decrease, SVM loss still 0

Cross-Entropy vs SVM Loss

$$L_i = -\log \frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Question: How to find weights that minimize these losses on our training data?

Answer: Optimization!

Today: Optimization

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:

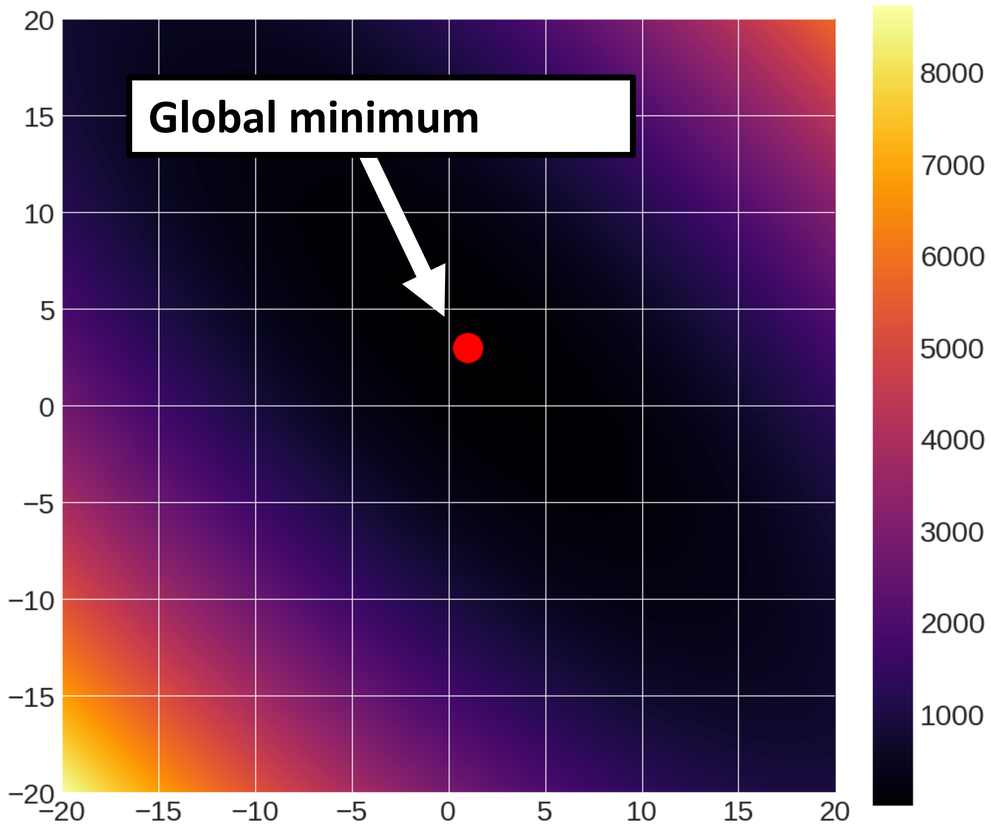
$$L(\boldsymbol{W}) = \boldsymbol{\lambda}\|\boldsymbol{W}\|_2^2 + \sum_{i=1}^{n} -\log\left(\frac{\exp((Wx)_{y_i})}{\sum_k \exp((Wx)_k))}\right)$$

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

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

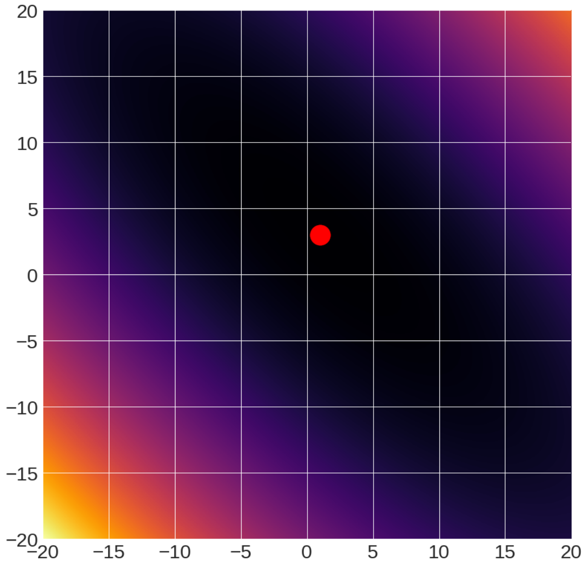
Sample Function to Optimize

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



Warning: This is 2D, intuition may not generalize to high dimension

Optimization: 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

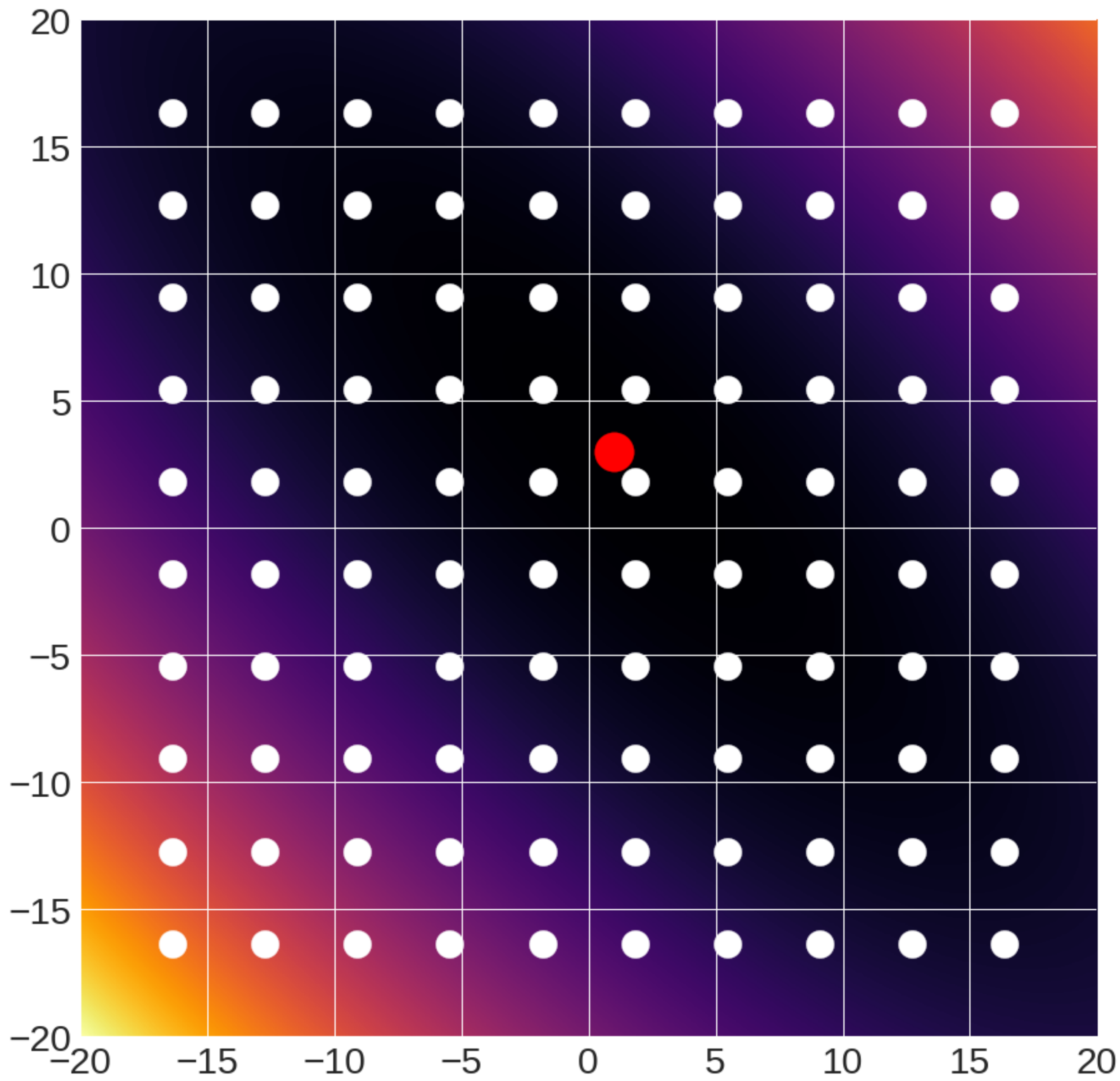
This image is CC0 1.0 public domain

Walking man image is CC0 1.0 public domain

Idea #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
```

Idea #1A: Grid Search



Idea #1A: Grid Search

Pros:

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

Cons:

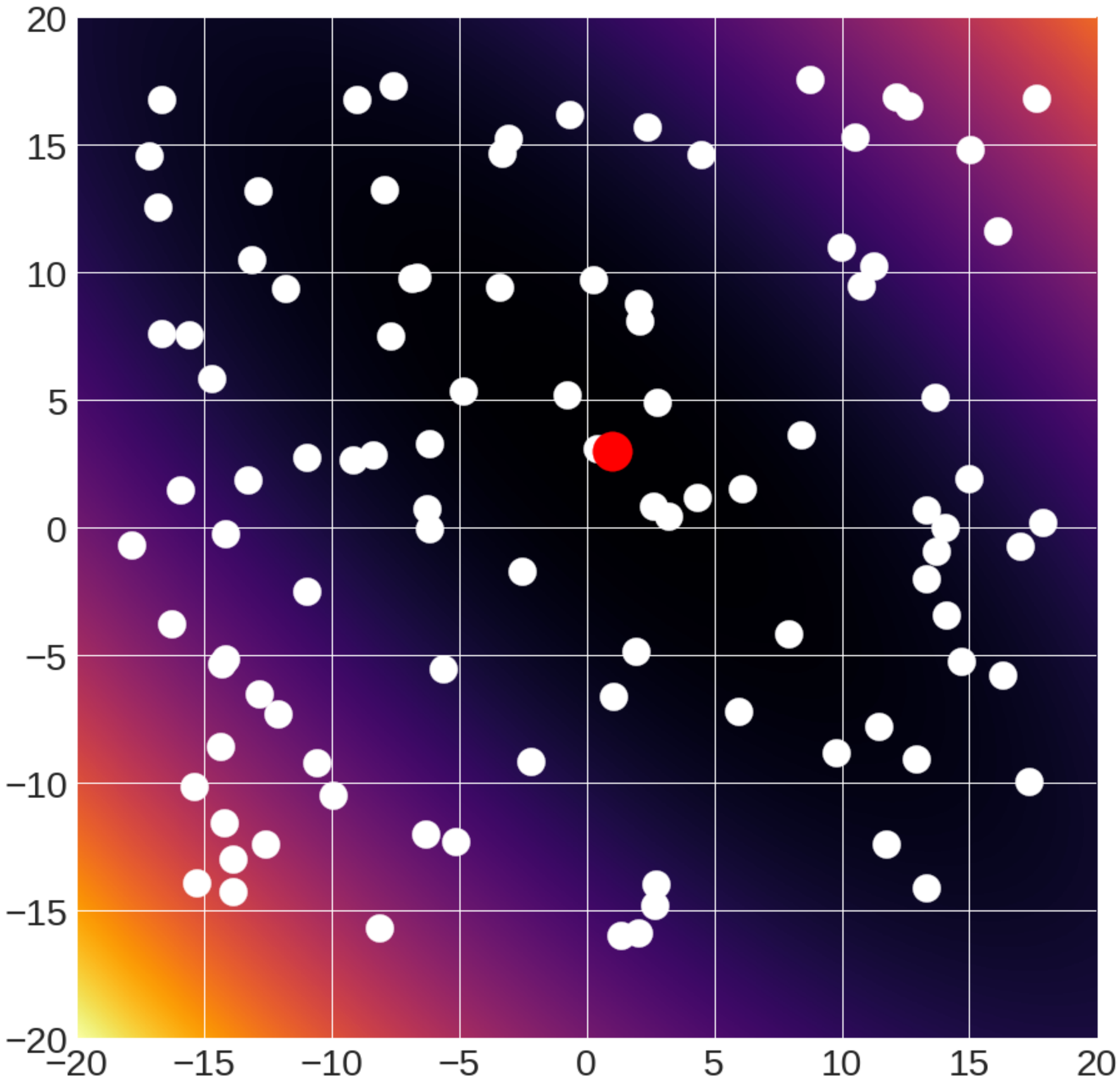
1. Scales horribly to high dimensional spaces

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(w) #evaluate
        if score < bestScore:
                best, bestScore = w, score
return best
```

Option #1B: Random Search



Option #1B: Random Search

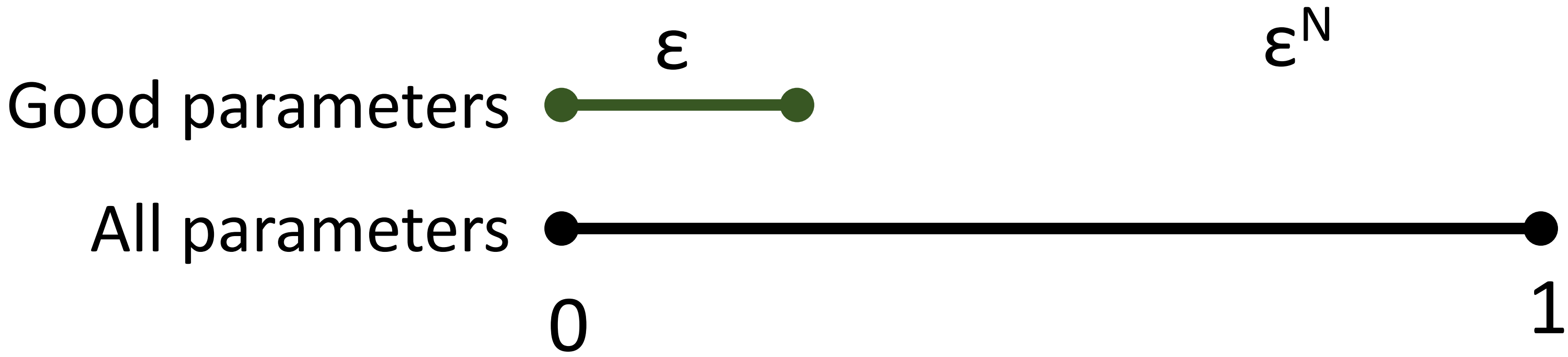
Pros:

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

Cons:

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

P(all correct) = ε^N



When To Use Options 1A / 1B?

Use these when

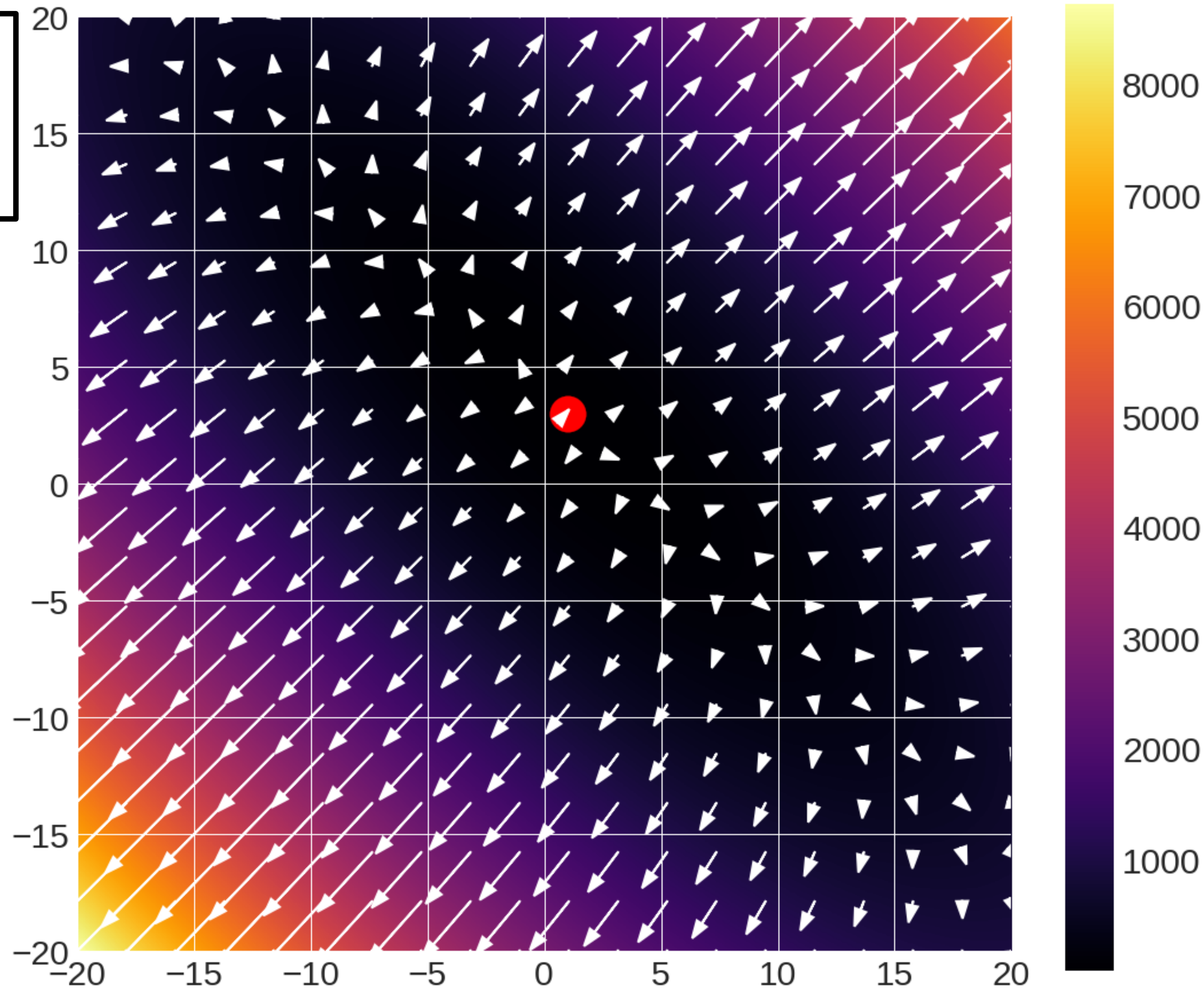
- Number of dimensions small, space bounded
- Objective is impossible to analyze (e.g., test accuracy if we use this distance function)

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

Idea #2: Follow the slope

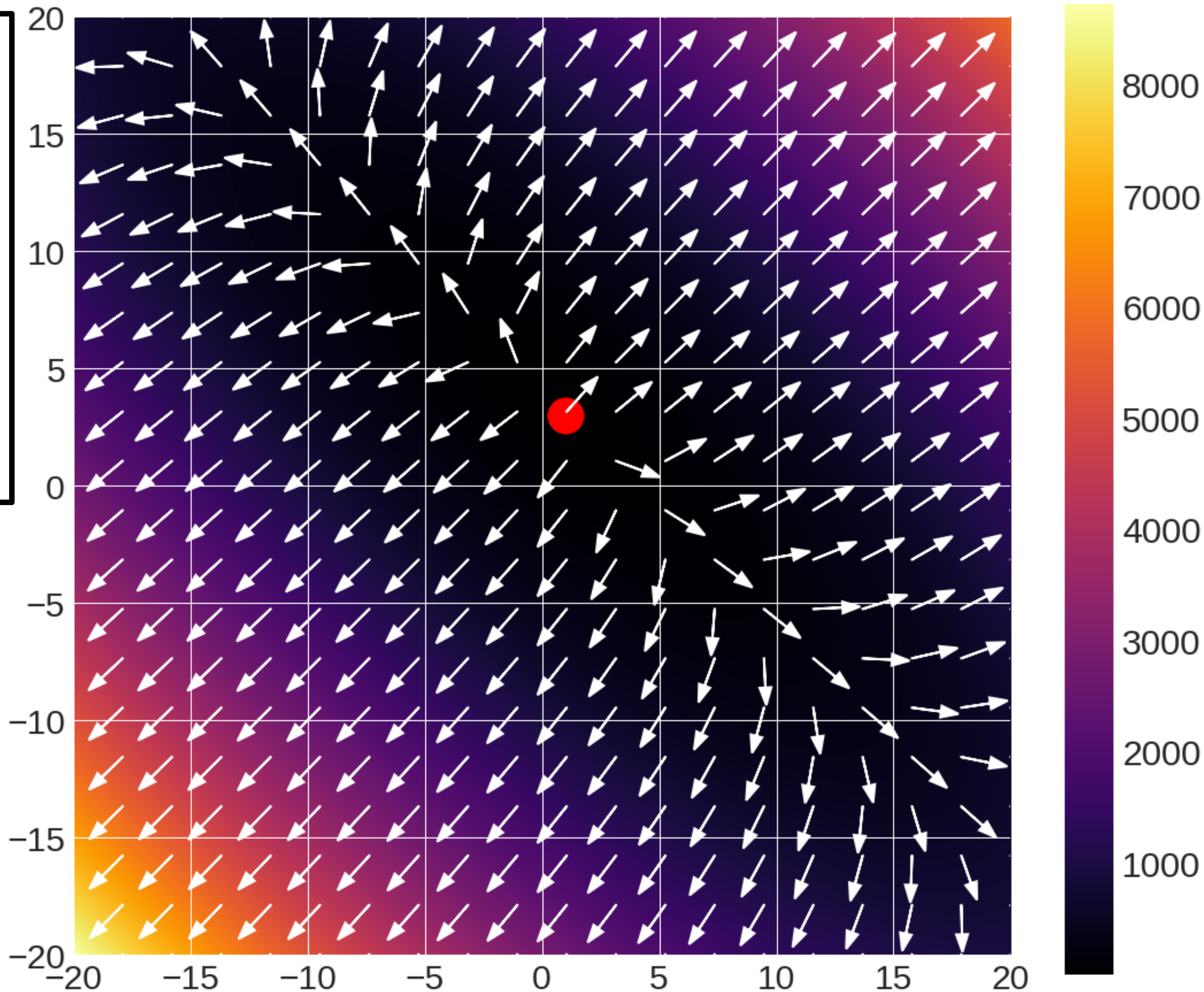
Idea #2: Follow the slope

Arrows:
gradient



Idea #2: Follow the slope

Arrows:
gradient direction
(scaled to unit length)



Idea #2: Follow the slope

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)?

$$L(\boldsymbol{w}) \quad \overset{\leq ?}{\underset{> ?}{}} \quad L(\boldsymbol{w} + \alpha \nabla_{\boldsymbol{w}} L(\boldsymbol{w}))$$

Idea #2: Follow the slope

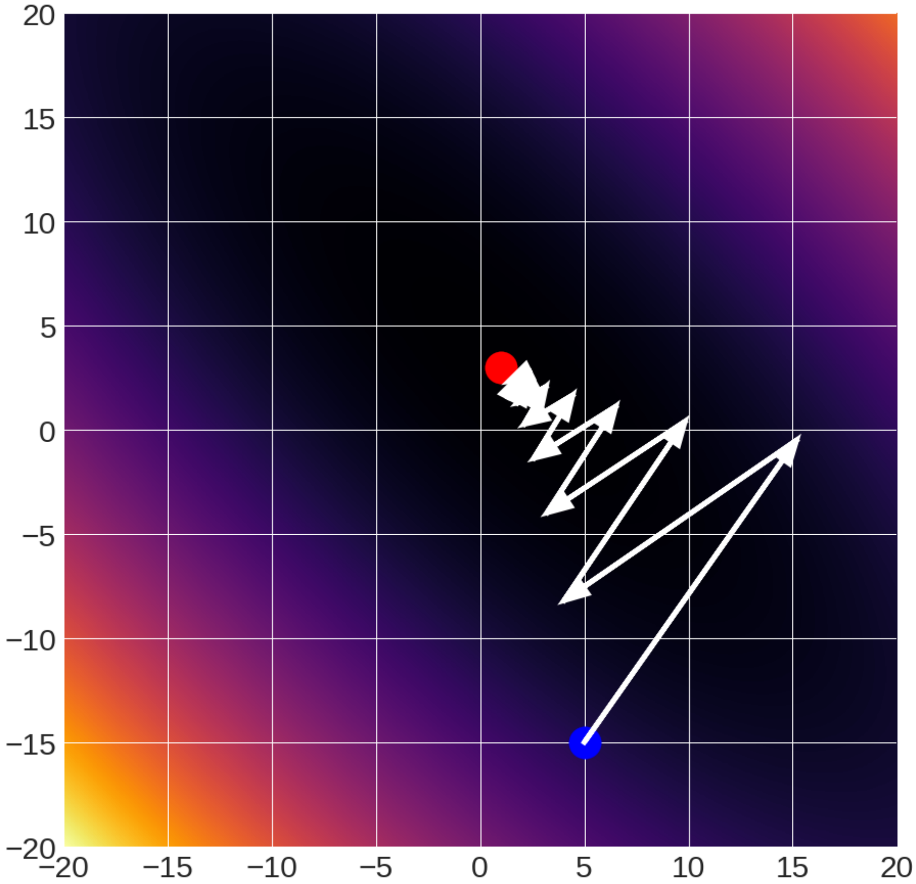
Method: at each step, move in direction of negative gradient

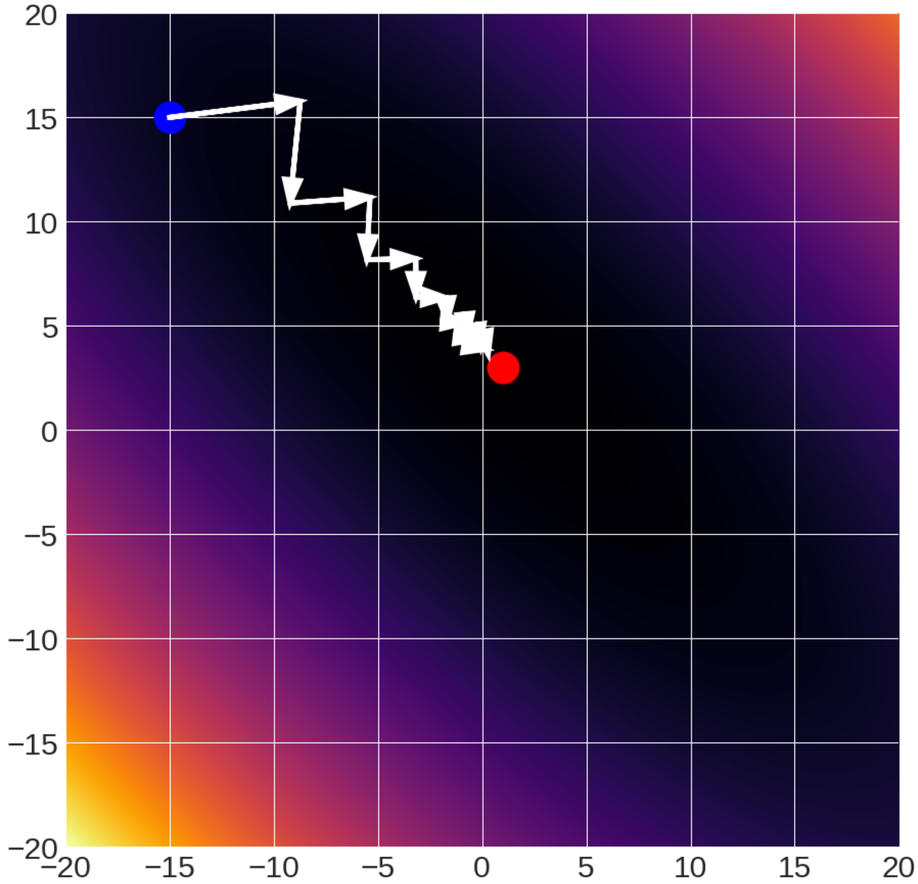
w0 = *initialize*() #initialize
for iter in range(numIters):
    g = $\nabla_{w} L(\boldsymbol{w})$ # eval gradient
    w = **w** + -*stepsize*(iter)***g** # update w
return **w**

Gradient Descent

Given starting point (blue)

$w_{i+1} = w_i + -9.8\mathrm{x}10^{-2}\ \mathrm{x}$ gradient





Computing Gradients: Numeric

How Do You Compute The Gradient?

Numerical Method:

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \begin{bmatrix} \boxed{\dfrac{\partial L(w)}{\partial w_1}} \\ \vdots \\ \dfrac{\partial L(w)}{\partial w_n} \end{bmatrix}$$

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 Gradients: Numeric

How Do You Compute The Gradient?

Numerical Method:

$$\nabla_w L(\boldsymbol{w}) = \begin{bmatrix} \boxed{\dfrac{\partial L(w)}{\partial x_1}} \\ \vdots \\ \dfrac{\partial L(w)}{\partial x_n} \end{bmatrix}$$

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

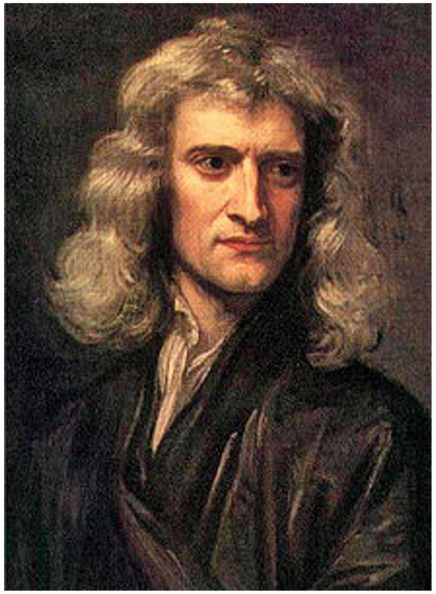
How many function evaluations per dimension?

Computing Gradients: Analytic

How Do You Compute The Gradient?

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

Better Idea: Use Calculus!



This image is in the public domain

This image is in the public domain

Computing Gradients: Analytic

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

$$\Big\downarrow \quad \frac{\partial}{\partial \boldsymbol{w}} \quad \Big\downarrow$$

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

Interpreting Gradients: 1 Sample

$$L(\boldsymbol{w}) = \lambda \|\boldsymbol{w}\|_2^2 + \left(y_i - \boldsymbol{w}^T \boldsymbol{x}_i\right)^2$$

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

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = 2\lambda \boldsymbol{w} + -(2(y - \boldsymbol{w}^T \boldsymbol{x})\boldsymbol{x})$$

$$-\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \overbrace{-2\lambda \boldsymbol{w}}^{\text{Push } \mathbf{w} \text{ towards } 0} + \underbrace{\overbrace{(2(y - \boldsymbol{w}^T \boldsymbol{x})}^{\alpha}\boldsymbol{x})}$$

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

Before: $w^T x$

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

Computing Gradients

- **Numeric gradient**: approximate, slow, easy to write
- **Analytic gradient**: exact, fast, error-prone

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check.**

```
torch.autograd.gradcheck(func, inputs, eps=1e-06, atol=1e-05, rtol=0.001, raise_exception=True, check_sparse_nnz=False, nondet_tol=0.0)
```

[SOURCE]

Check gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` that are of floating point type and with `requires_grad=True`.

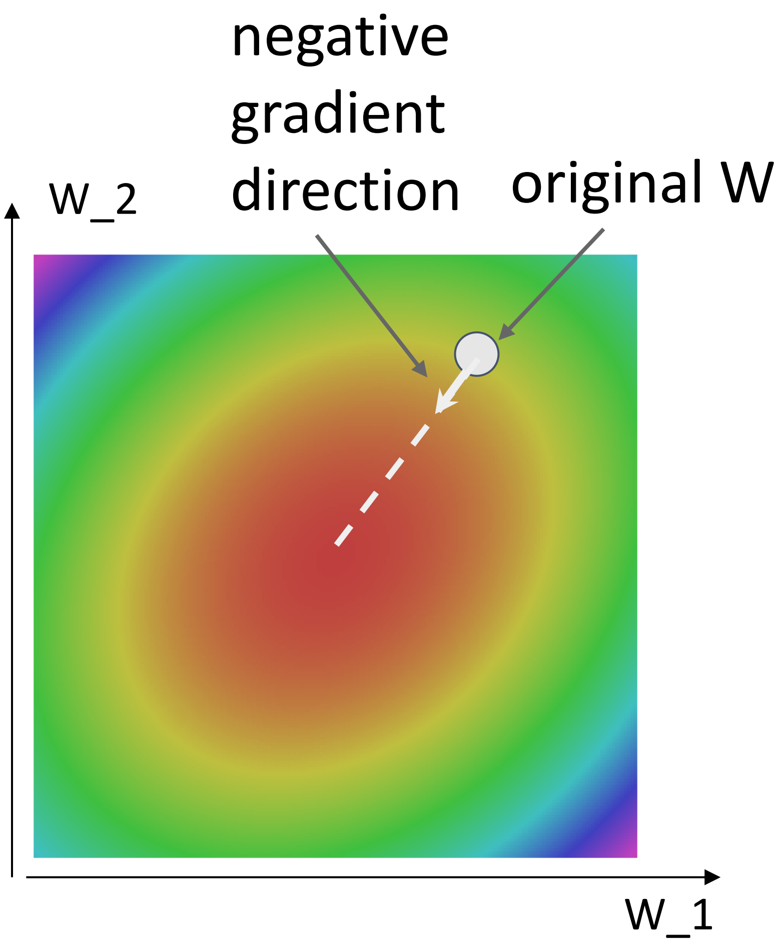
The check between numerical and analytical gradients uses `allclose()`.

Gradient Descent

Iteratively step in the direction of the negative gradient
(direction of local steepest descent)

```python
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:
- Weight initialization method
- Number of steps
- Learning rate



Batch Gradient Descent

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

Problem: Full sum is expensive when N is large!

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Solution: Approximate sum using a **minibatch** of examples, e.g. 32

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

Problem: Full sum is expensive when N is large!

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Solution: Approximate sum using a **minibatch** of examples, e.g. 32

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

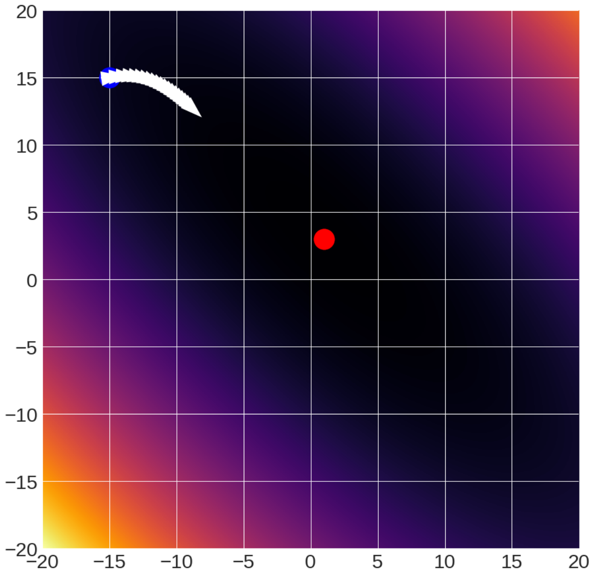
Hyperparameters:
- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

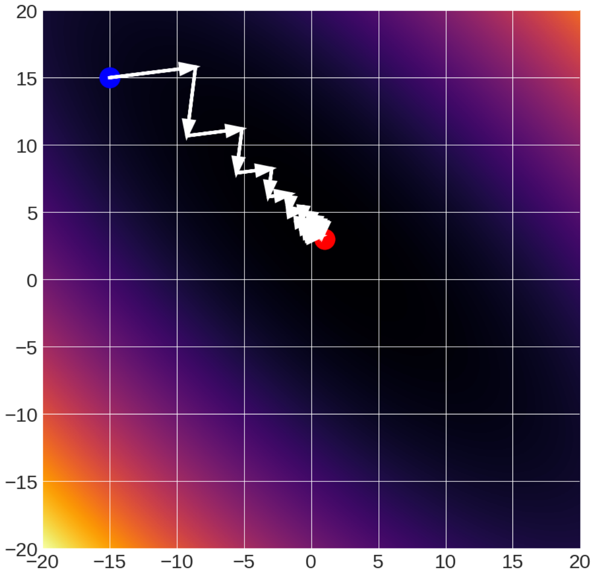
Note: Some people say "stochastic gradient descent" is batch size 1, and "minibatch gradient descent" for other batch sizes. I think this distinction is confusing, and use "stochastic gradient descent" for any minibatch size

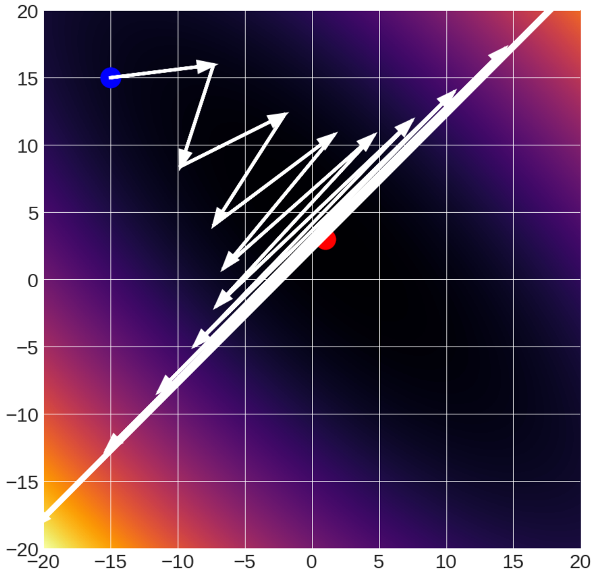
Gradient Descent: Learning Rate

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

$1x10^{-2}$	$10x10^{-2}$	$12x10^{-2}$
falls short	converges	diverges

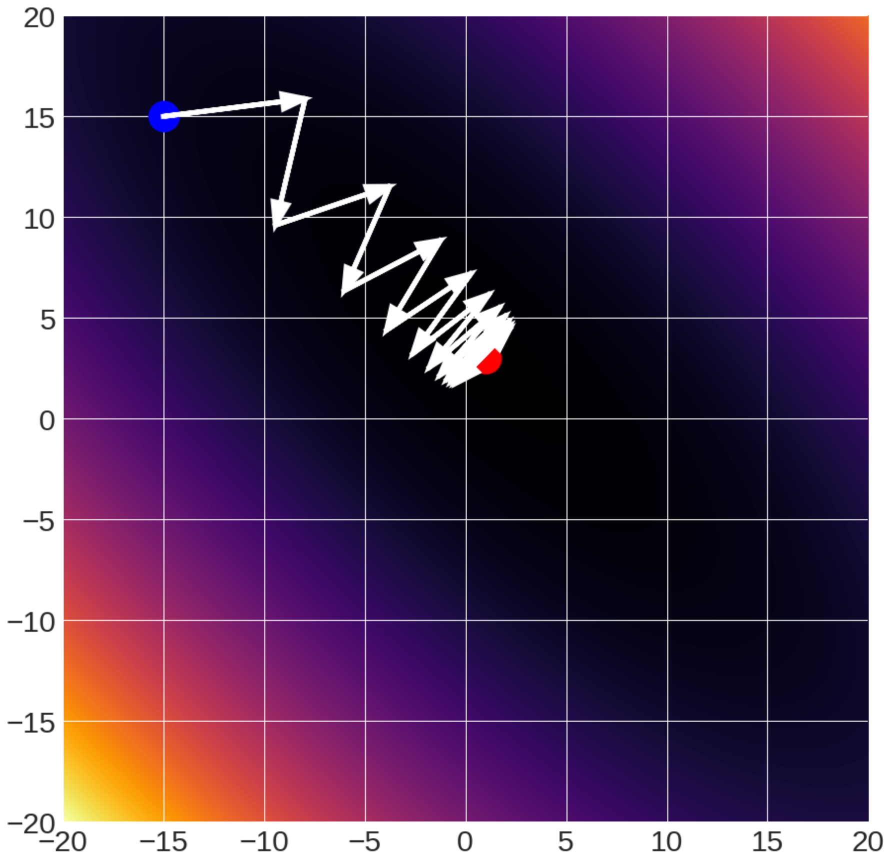






Gradient Descent: Learning Rate

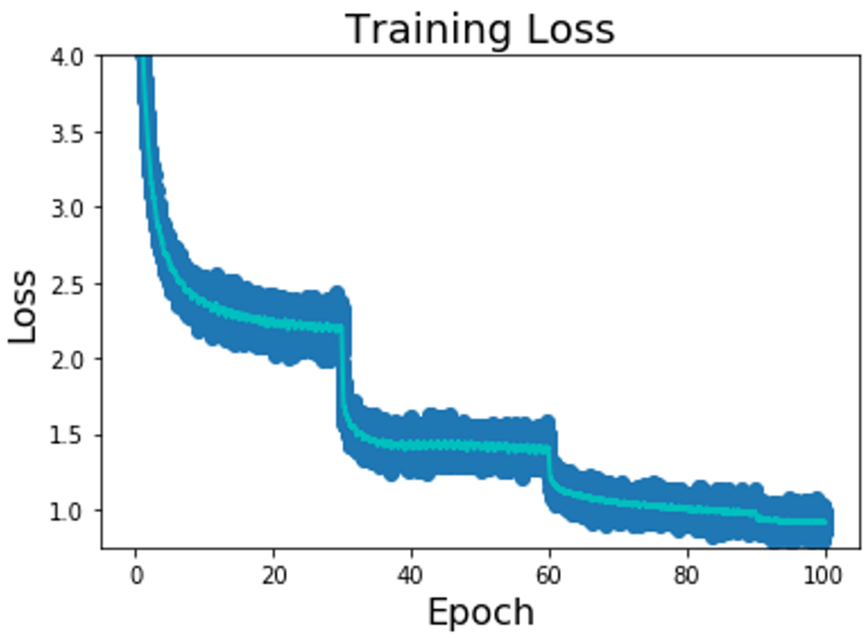
11x10^{-2} :oscillates
(Raw gradients)

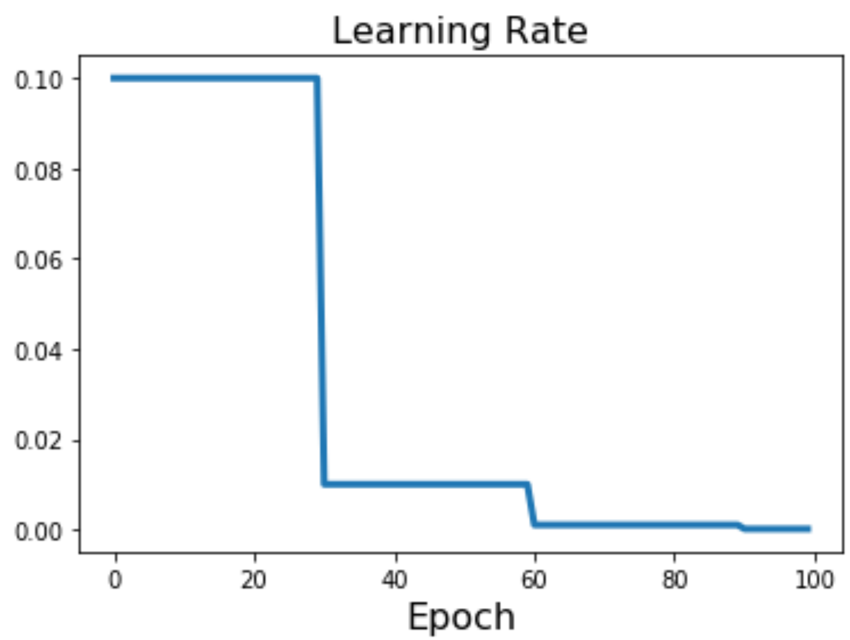


Learning Rate Decay

Idea: Start with high learning rate, reduce it over time.

Step Decay: Reduce by some factor at fixed iterations

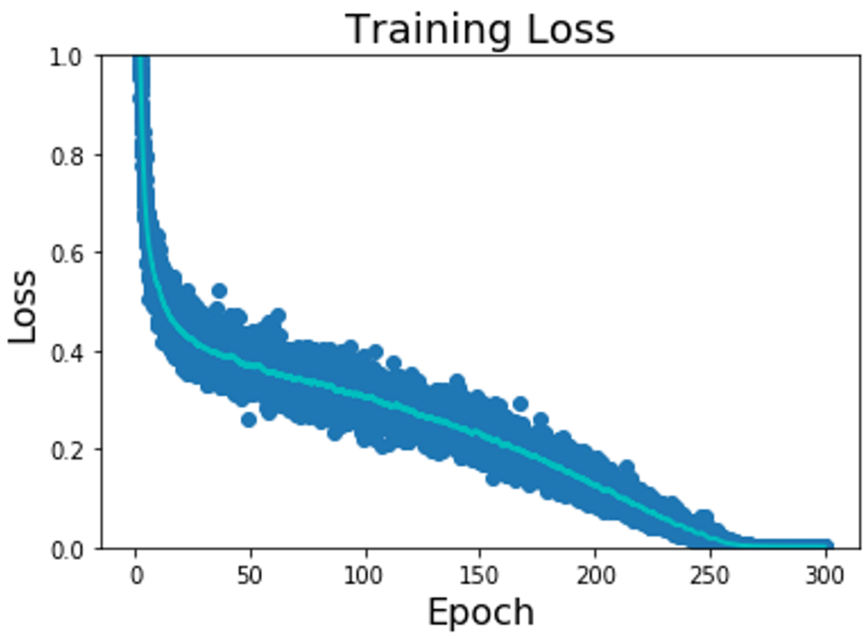


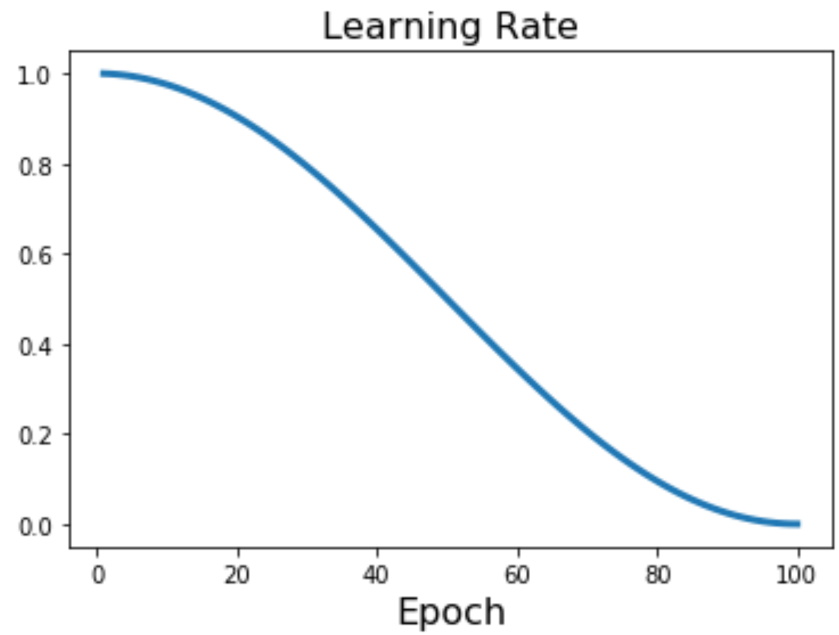


Learning Rate Decay

Idea: Start with high learning rate, reduce it over time.

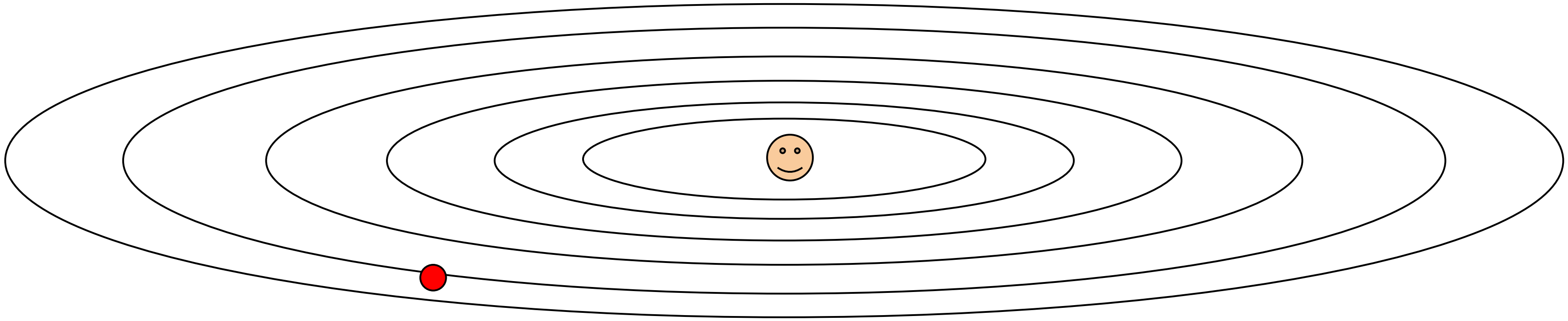
Cosine Decay: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + cos\left(\frac{t\pi}{T}\right)\right)$





Problems with SGD

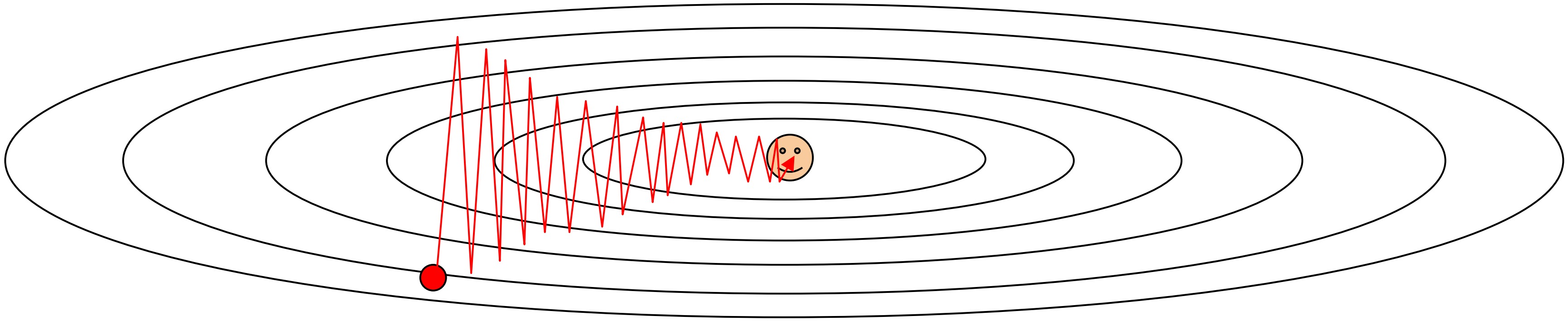
What if loss changes quickly in one direction and slowly in another?



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with SGD

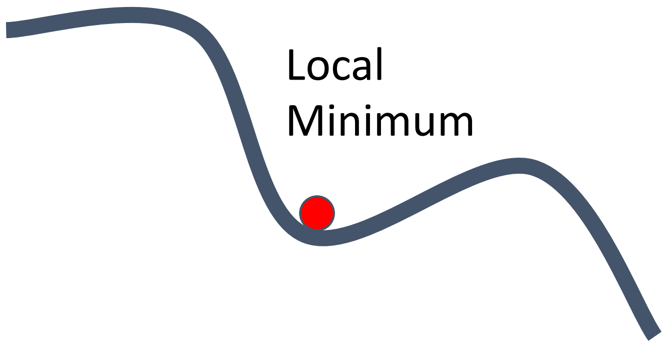
What if loss changes quickly in one direction and slowly in another?
Slow progress along shallow dimension, jitter along steep direction



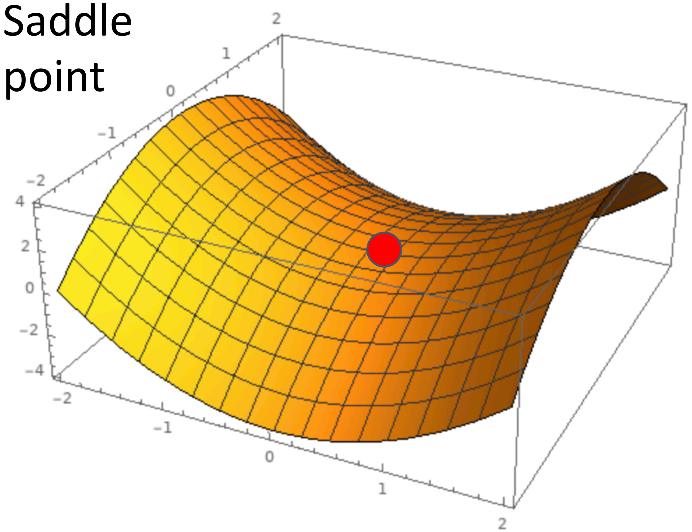
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with SGD

What if the loss function has a **local minimum** or **saddle point**?



Gradient is zero,
SGD gets stuck



Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

SGD

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD + Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically $\rho = 0.9$ or 0.99

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

SGD + Momentum

SGD + Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v - learning_rate * dw
  w += v
```

SGD + Momentum

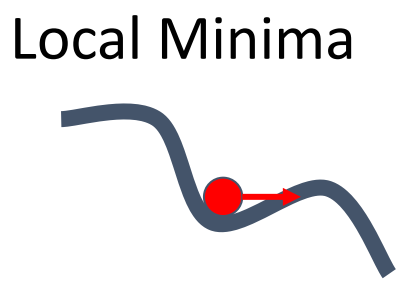
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

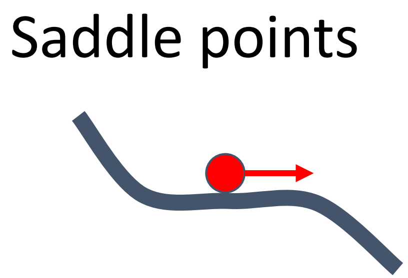
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

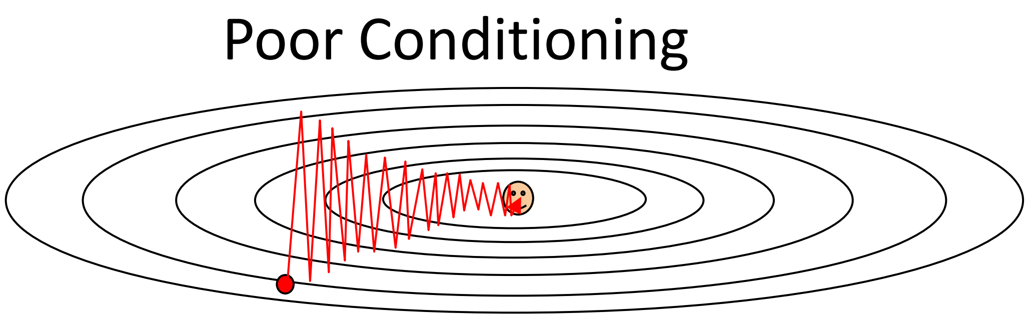
You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

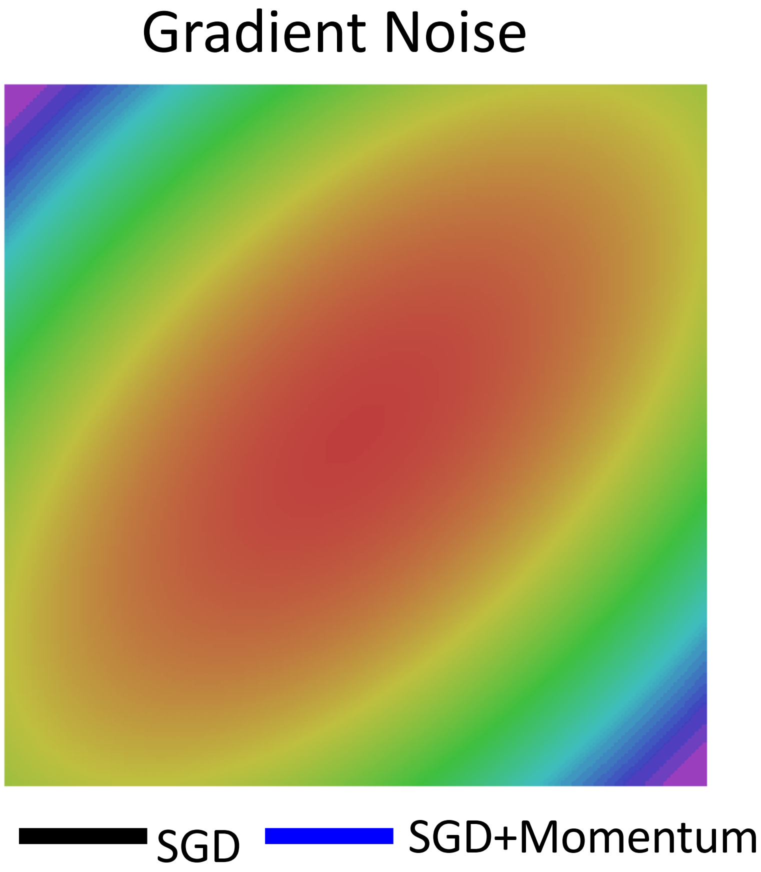
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

SGD + Momentum









Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Other Update Rules: Adam

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Adam with beta1 = 0.9,
beta2 = 0.999, and learning_rate = 1e-3, 5e-4, 1e-4
is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam: Very Common in Practice!

for input to the CNN; each colored pixel in the image yields a 7D one-hot vector. Following common practice, the network is trained end-to-end using stochastic gradient descent with the Adam optimizer [22]. We anneal the learning rate to 0 using a half cosine schedule without restarts [28].

Bakhtin, van der Maaten, Johnson, Gustafson, and Girshick, NeurIPS 2019

We train all models using Adam [23] with learning rate 10^{-4} and batch size 32 for 1 million iterations; training takes about 3 days on a single Tesla P100. For each mini-batch we first update f, then update D_{img} and D_{obj}.

Johnson, Gupta, and Fei-Fei, CVPR 2018

ganized into three residual blocks. We train for 25 epochs using Adam [27] with learning rate 10^{-4} and 32 images per batch on 8 Tesla V100 GPUs. We set the `cubify` thresh-

Gkioxari, Malik, and Johnson, ICCV 2019

sampled with each bit drawn uniformly at random. For gradient descent, we use Adam [29] with a learning rate of 10^{-3} and default hyperparameters. All models are trained with batch size 12. Models are trained for 200 epochs, or 400 epochs if being trained on multiple noise layers.

Zhu, Kaplan, Johnson, and Fei-Fei, ECCV 2018

16 dimensional vectors. We iteratively train the Generator and Discriminator with a batch size of 64 for 200 epochs using Adam [22] with an initial learning rate of 0.001.

Gupta, Johnson, et al, CVPR 2018

Adam with beta1 = 0.9,
beta2 = 0.999, and learning_rate = 1e-3, 5e-4, 1e-4
is a great starting point for many models!

Optimization in Practice

- **Conventional wisdom**: minibatch stochastic gradient descent (SGD) + momentum (package implements it for you) + some sensibly changing learning rate
- The above is typically what is meant by "SGD"
- Other update rules exist (Adam very common); sometimes better, sometimes worse than SGD

Optimizing Everything

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

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

- 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

Next Time:
Nonlinear Models,
Neural Networks!