中山大學
SUN YAT-SEN UNIVERSITY

Lecture 17: Regularization and Optimization

Pattern Recognition and Computer Vision

Guanbin Li,
School of Computer Science and Engineering, Sun Yat-Sen University

扫码签到

Image Classification

- A core task in Computer Vision

This image by Nikita is licensed under CC-BY 2.0

(assume given a set of labels)
{dog, cat, truck, plane, ...}

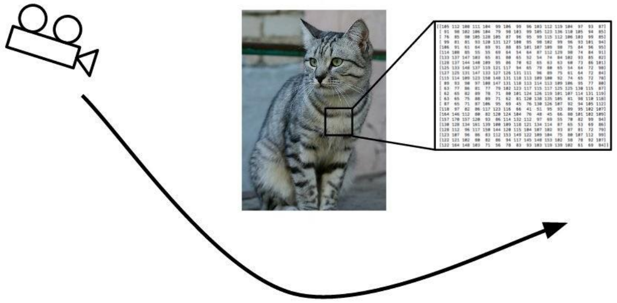
$\longrightarrow$

cat
dog
bird
deer
truck

Image Classification

- **Recall from last time**: Challenges of recognition

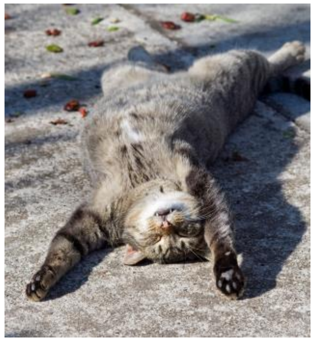
Viewpoint



Illumination

This image is CC0 1.0 public domain

Deformation



This image by Umberto Salvagnin is licensed under CC-BY 2.0

Occlusion

This image by jonsson is licensed under CC-BY 2.0

This image is CC0 1.0 public domain

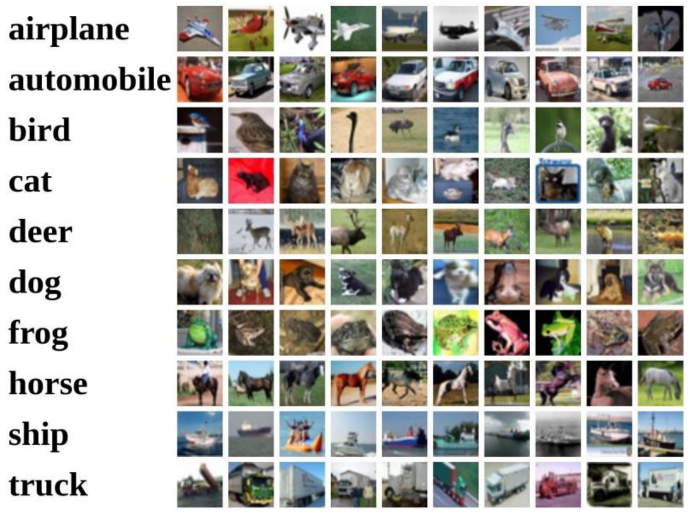
Clutter

This image is CC0 1.0 public domain

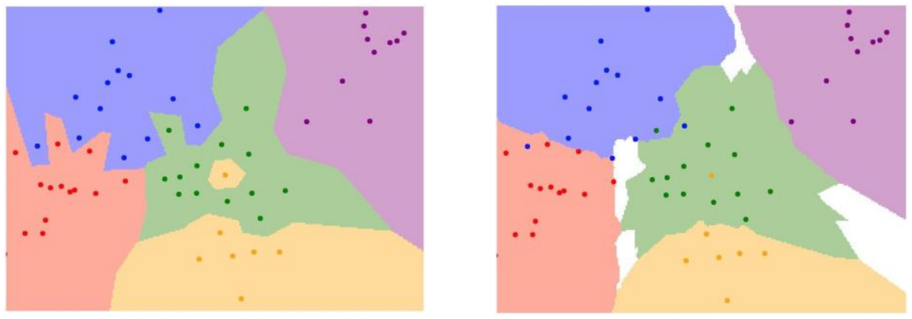
Intraclass Variation

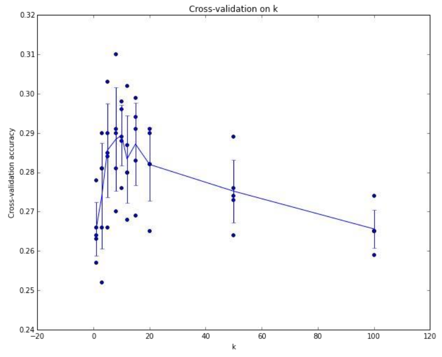
Image Classification

- **Recall from last time**： data-driven approach， kNN



1-NN classifier 5-NN classifier





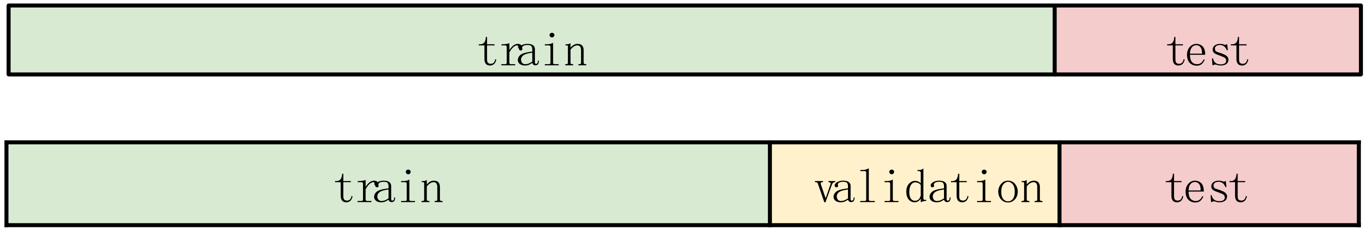


Image Classification

- **Recall from last time**: Linear Classifier

Image

Array of **32x32x3** numbers (3072 numbers total)

→ f(**x**,**W**) → **10** numbers giving class scores

↑

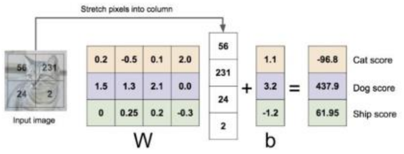
W

parameters or weights

$$f(x;W) = Wx + b$$

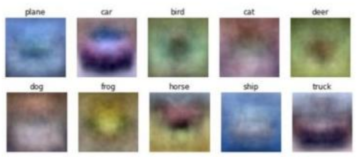
Algebraic Viewpoint

$f(x,W) = Wx$



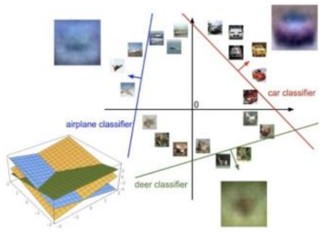
Visual Viewpoint

One template per class



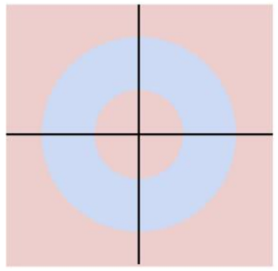
Geometric Viewpoint

Hyperplanes cutting up space



Class 1:
1 <= L2 norm <= 2

Class 2:
Everything else



Class 1:
Three modes

Class 2:
Everything else

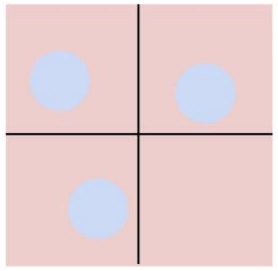
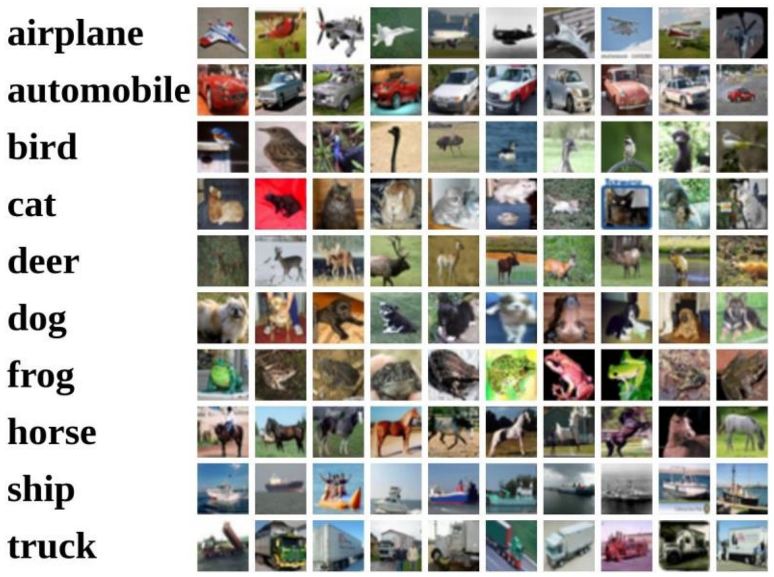
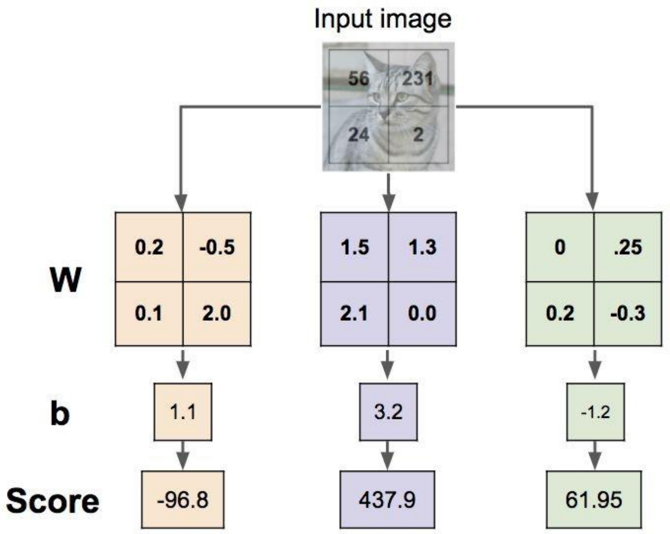


Image Classification

- Interpreting a Linear Classifier: Visual Viewpoint





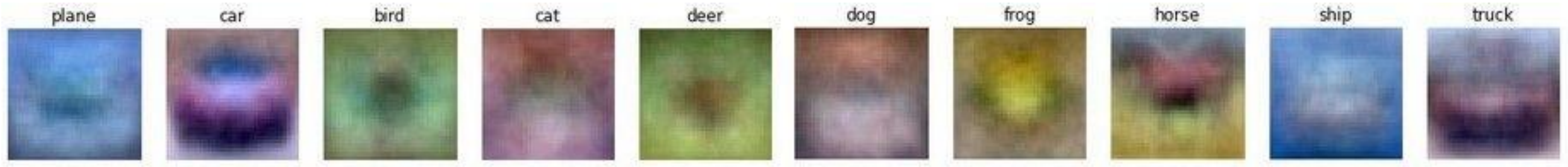
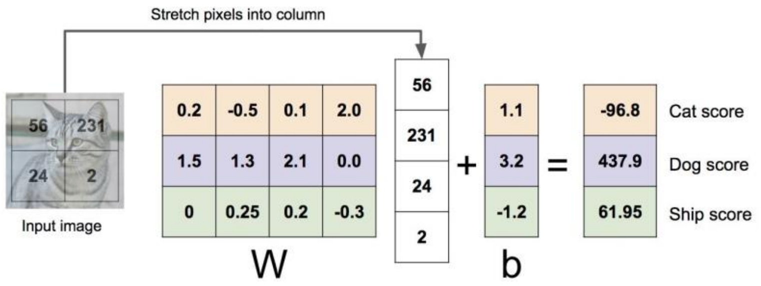


Image Classification

- Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

Algebraic Viewpoint

$f(x,W) = Wx$



Input image

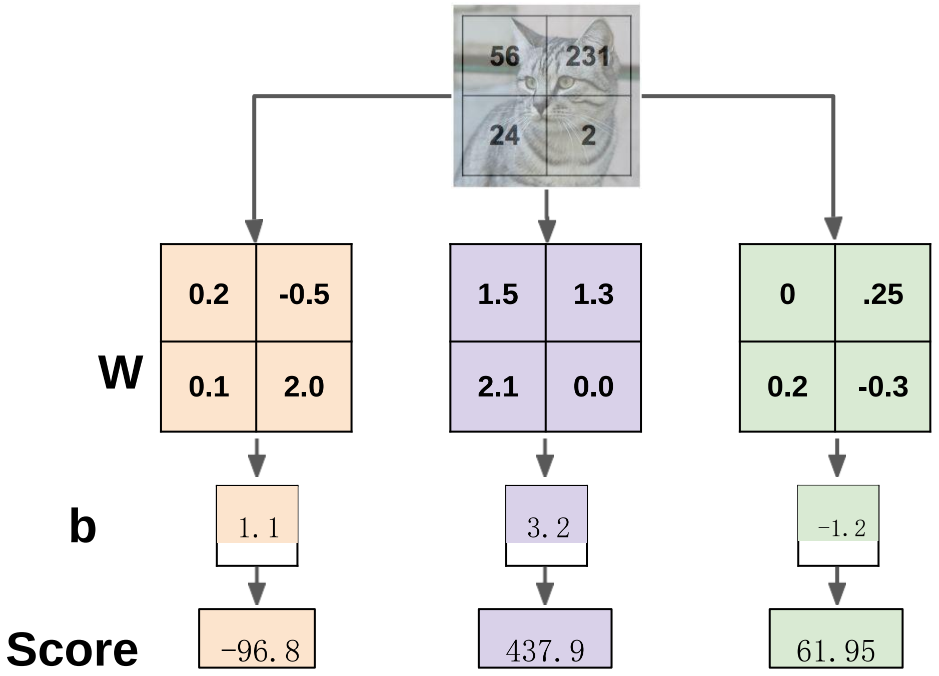
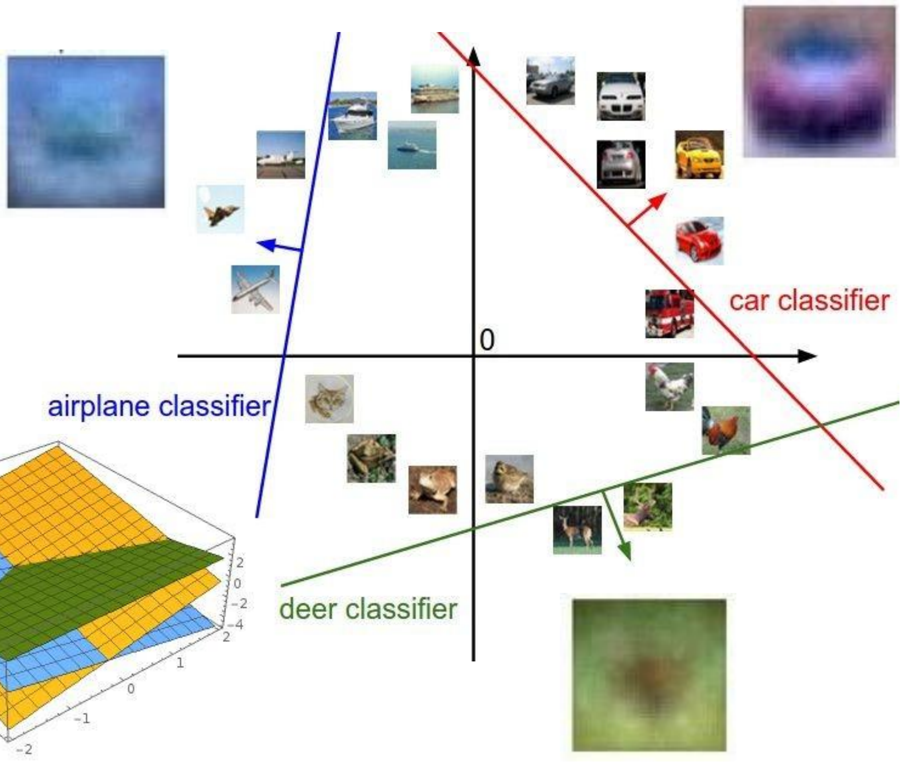
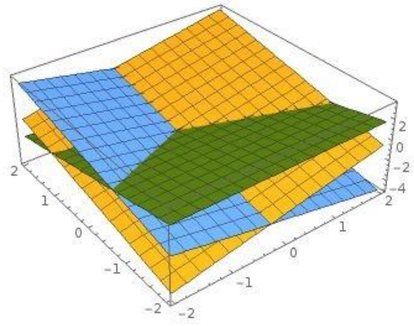


Image Classification

- Interpreting a Linear Classifier: Geometric Viewpoint





Plot created using Wolfram Cloud

$f(x,W) = Wx + b$

Array of **32x32x3** numbers (3072 numbers total)

Image Classification

- Suppose: 3 training examples, 3 classes. With some W the scores are: $f(x, W) = Wx$

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

A **loss function** tells how good our current classifier is.

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^{N}$$

Where x_i is image and y_i is (integer) label

Loss over the dataset is a average
loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Image Classification

Softmax vs. SVM $L_i = -\log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$ $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$

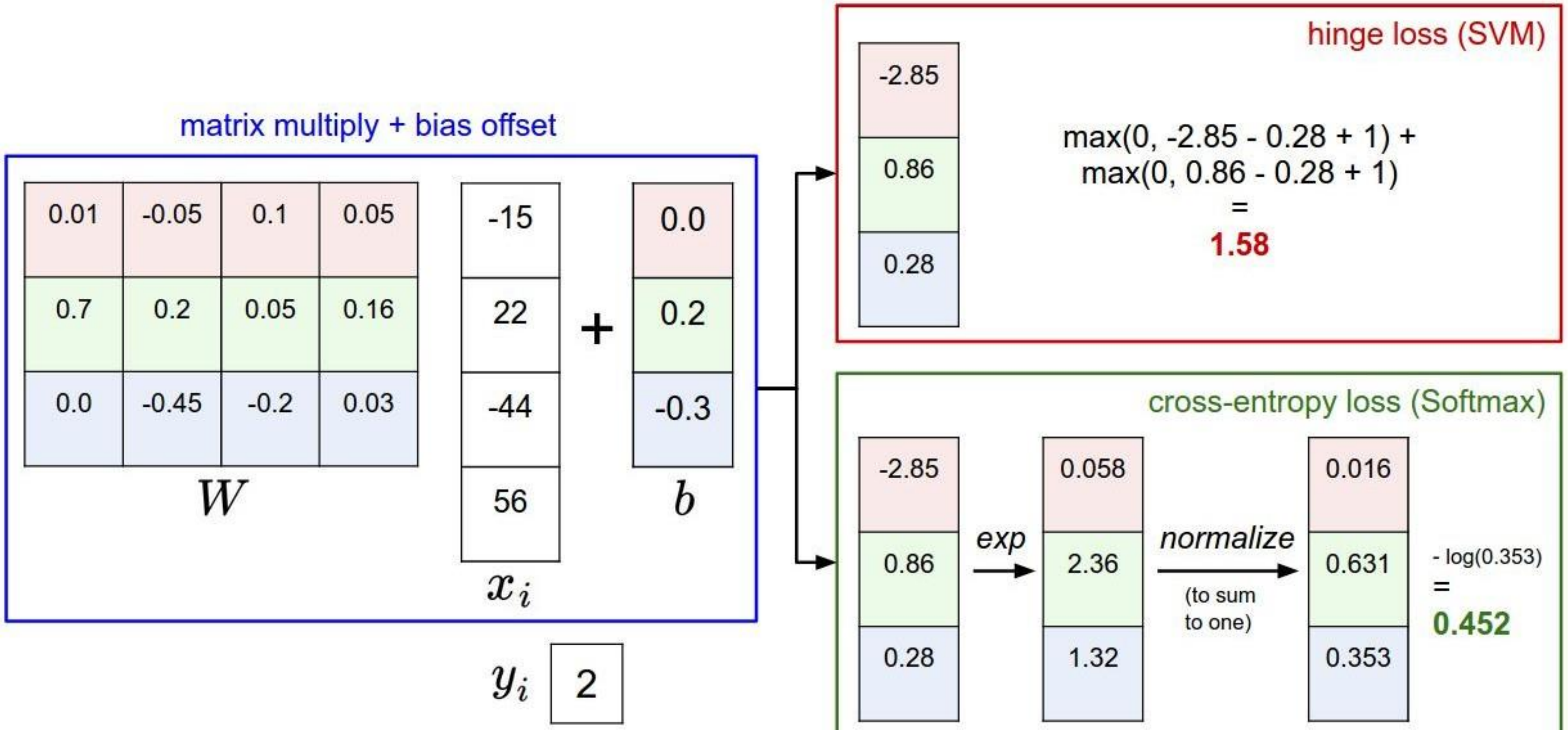


Image Classification

$$f(x, W) = Wx$$

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

Q: Suppose that we found a W such that L = 0. Is this W unique?

Image Classification

$$f(x, W) = Wx$$

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

E. g. Suppose that we found a W such that L = 0. Is this W unique?

No! 2W is also has L = 0!

Image Classification

- Suppose: 3 training examples, 3 classes. With some W the scores are: $f(x, W) = Wx$

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

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

Before:

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

With W twice as large:

$= \max(0, 2.6 - 9.8 + 1)$
$+\max(0, 4.0 - 9.8 + 1)$
$= \max(0, -6.2) + \max(0, -4.8)$
$= 0 + 0$
$= 0$

Image Classification

$$f(x, W) = Wx$$

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

E. g.　Suppose that we found a W such that L = 0.　Is this W unique?

No! 2W is also has L = 0!
How do we choose between W and 2W?z

Regularization

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}$$

Data loss: Model predictions should match training data

Regularization

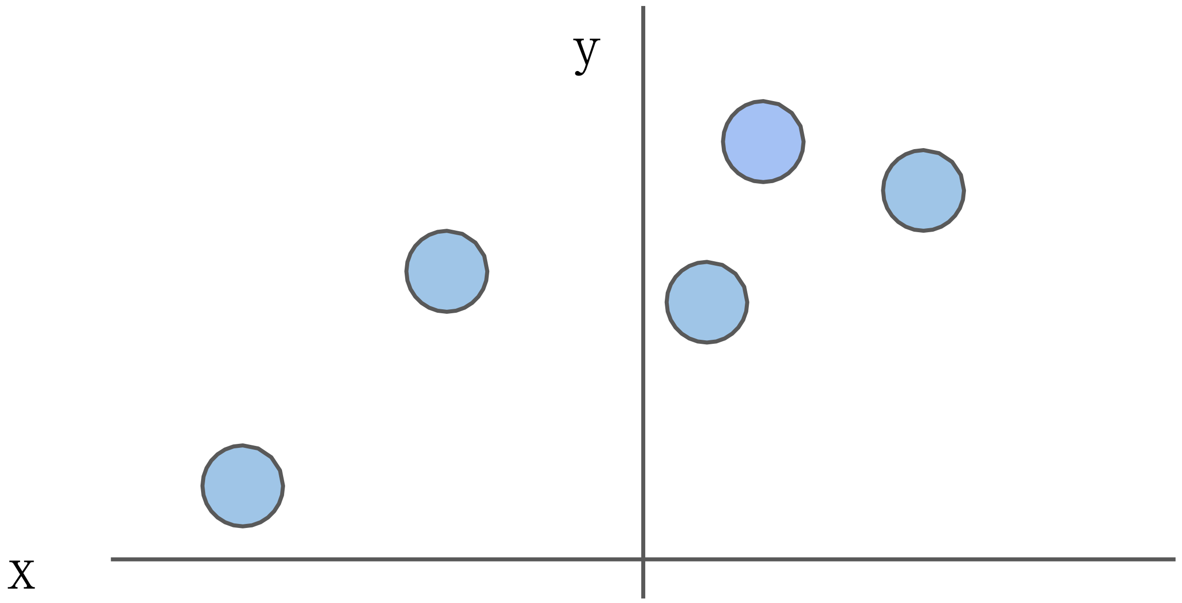
$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing ***too*** well on training data

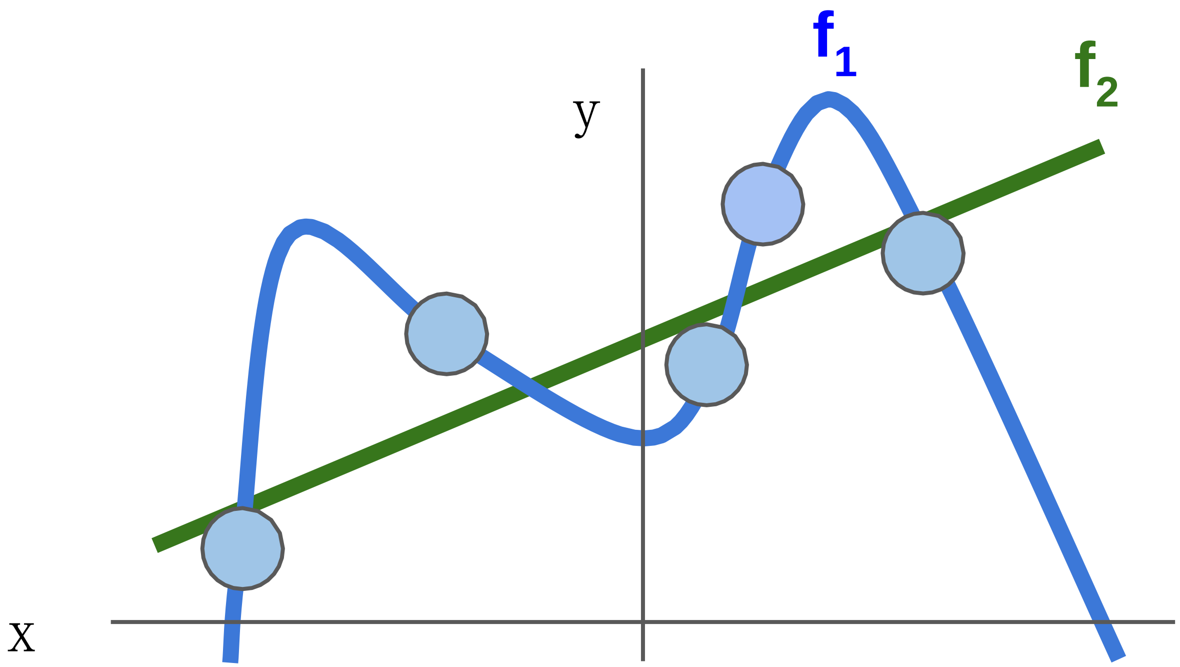
Regularization

- Regularization intuition: toy example training data



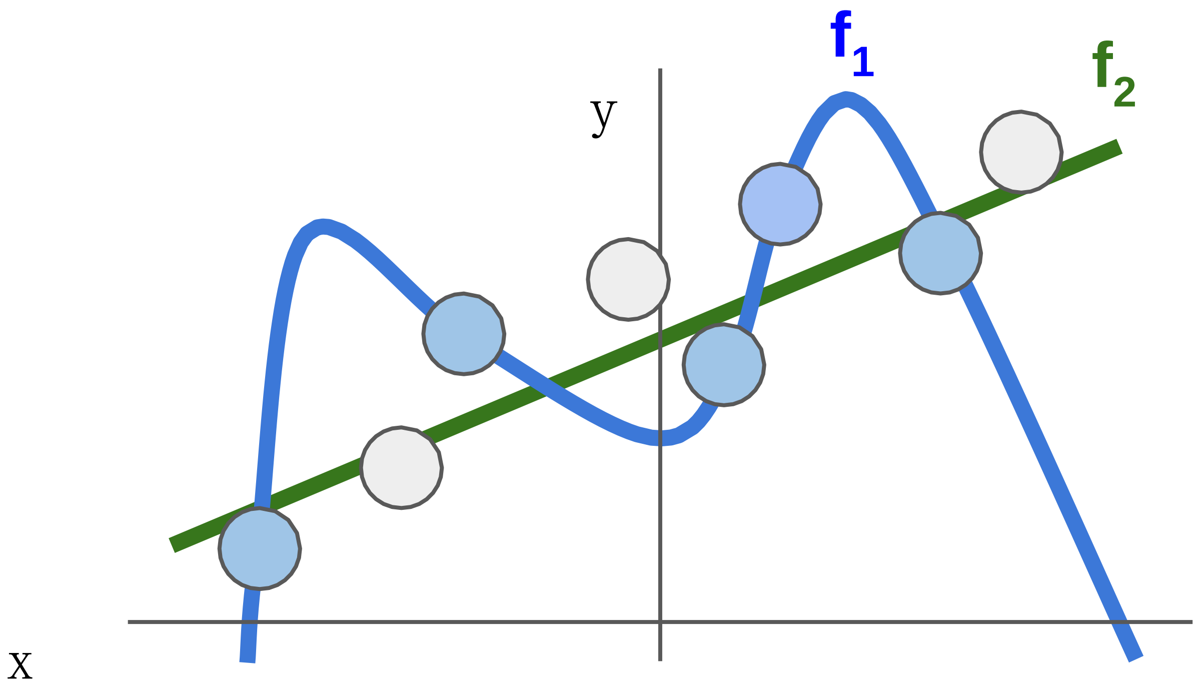
Regularization

- Regularization intuition: Prefer Simpler Models



Regularization

- Regularization: Prefer Simpler Models



Regularization pushes against fitting the data *too* well so we don’t fit noise in the data

Regularization

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

Data loss：Model predictions should match training data

Regularization：Prevent the model from doing *too* well on training data

Occam's Razar：Among multiple competing hypotheses, the simplest is the best， William of Ockham 1285-1347

Regularization

λ = regularization strength (hyperparameter)

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

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Regularization

λ = regularization strength (hyperparameter)

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

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1+L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

<u>L2 regularization:</u> $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1+L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

More complex:

Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

Regularization

- Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will the L2 regularizer prefer?

Regularization

- Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = \boxed{[0.25, 0.25, 0.25, 0.25]}$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will the L2 regularizer prefer?

L2 regularization likes to “spread out ” the weights

Regularization

- Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = \boxed{[0.25, 0.25, 0.25, 0.25]}$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will the L2 regularizer prefer?

L2 regularization likes to "spread out" the weights

Which one would L1 regularization prefer?

Regularization

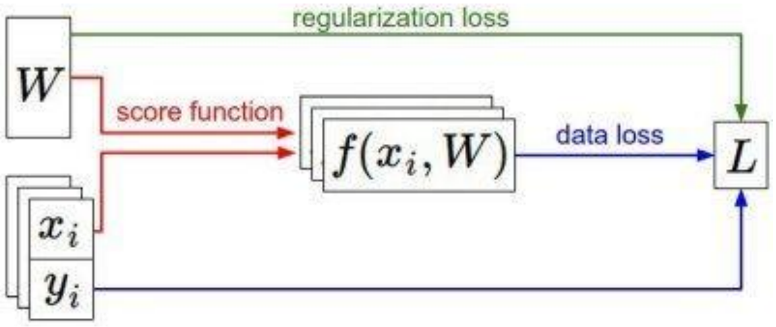
Recap

- We have some dataset of (x, y)
- We have a **score function:** $s = f(x; W) \overset{e.g.}{=} Wx$
- We have a **loss function**:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$ Softmax

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

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$$ Full loss



Regularization

Recap

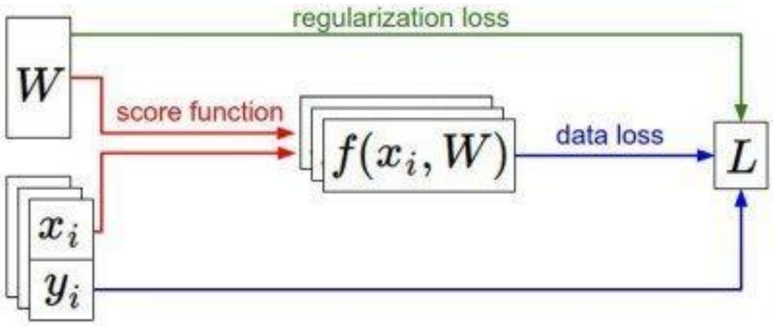
How do we find the best W?

- We have some dataset of (x, y)
- We have a **score function:** $s = f(x;W) \overset{e.g}{=} Wx$
- We have a **loss function**:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$ Softmax

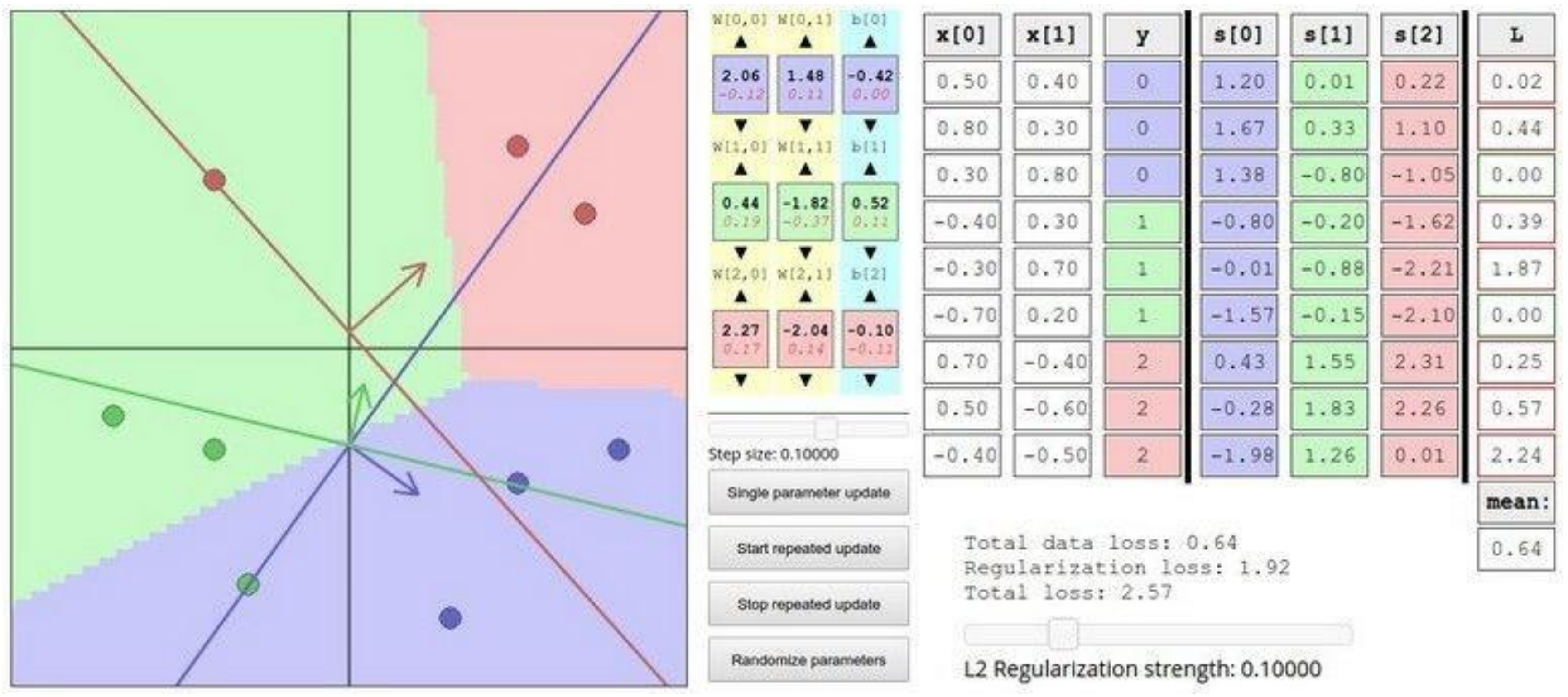
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$ SVM

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$ Full loss



Regularization

- Interactive Web Demo



W[0,0]	W[0,1]	b[0]
2.06 -0.12	1.48 0.11	-0.42 0.00
W[1,0]	**W[1,1]**	**b[1]**
0.44 0.19	-1.82 -0.37	0.52 0.11
W[2,0]	**W[2,1]**	**b[2]**
2.27 0.27	-2.04 0.14	-0.10 -0.11

Step size: 0.10000

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	1.20	0.01	0.22	0.02
0.80	0.30	0	1.67	0.33	1.10	0.44
0.30	0.80	0	1.38	-0.80	-1.05	0.00
-0.40	0.30	1	-0.80	-0.20	-1.62	0.39
-0.30	0.70	1	-0.01	-0.88	-2.21	1.87
-0.70	0.20	1	-1.57	-0.15	-2.10	0.00
0.70	-0.40	2	0.43	1.55	2.31	0.25
0.50	-0.60	2	-0.28	1.83	2.26	0.57
-0.40	-0.50	2	-1.98	1.26	0.01	2.24
						mean:
						0.64

```
Total data loss: 0.64
Regularization loss: 1.92
Total loss: 2.57
```

L2 Regularization strength: 0.10000

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

Optimization

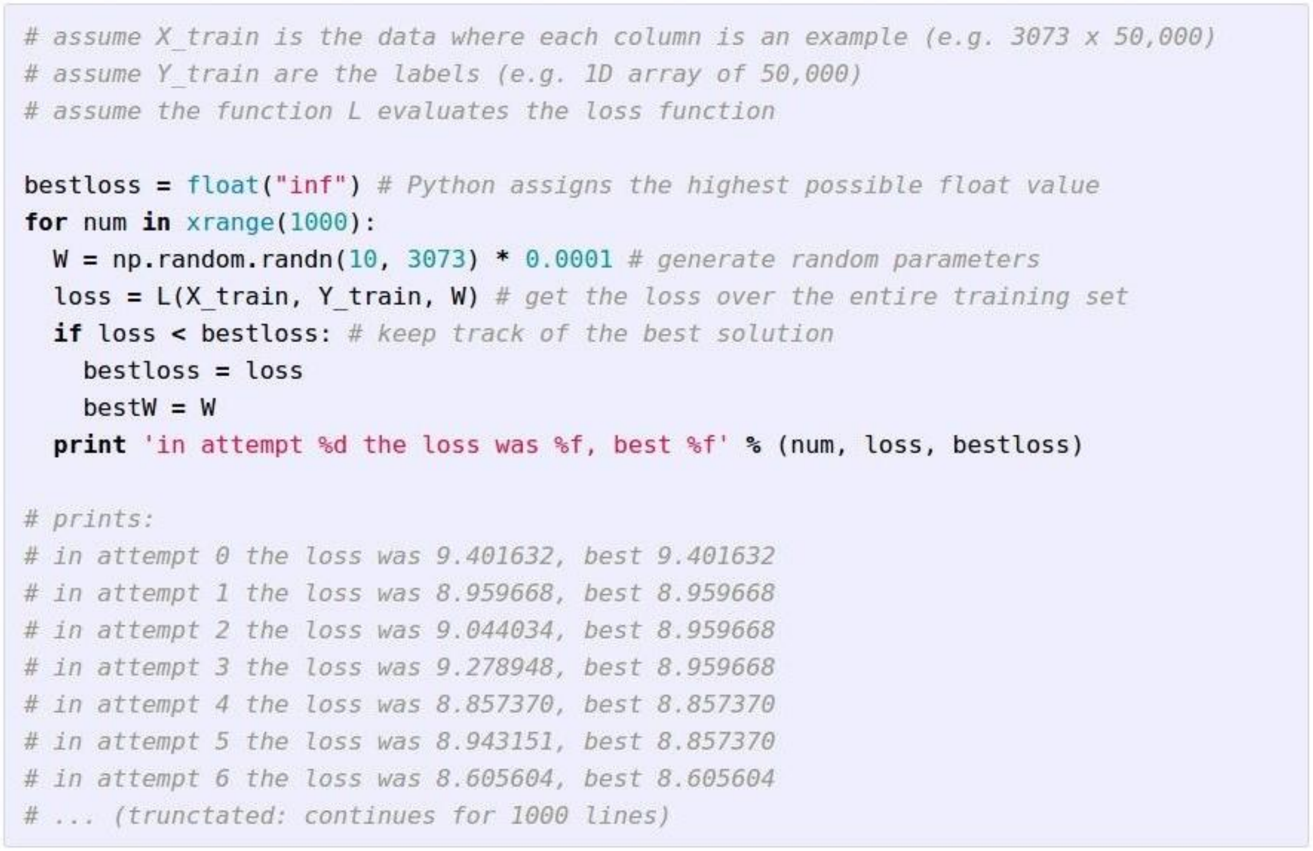
Optimization

Optimization

Walking man image is CC0 1.0 public domain

Optimization

- Strategy #1: A first very bad idea solution: **Random search**



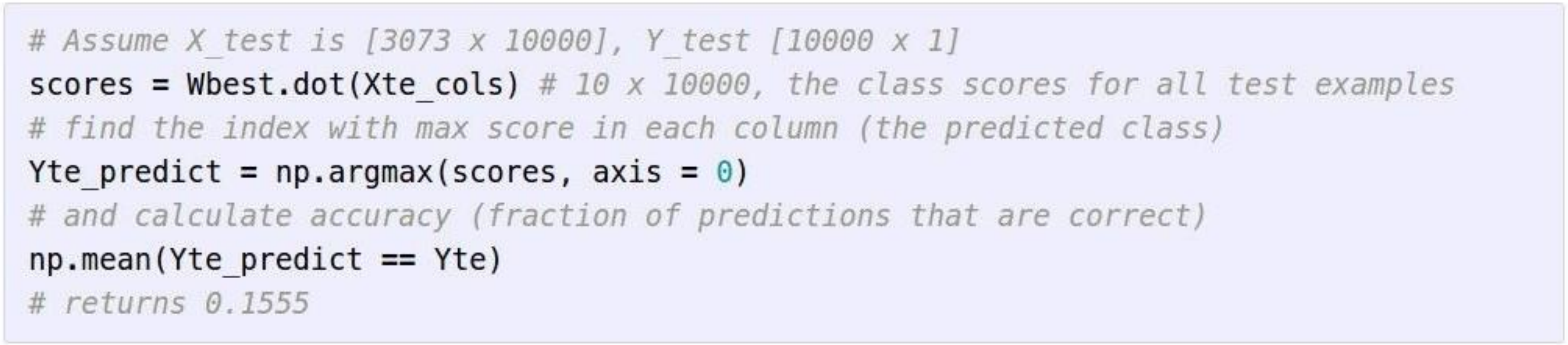
```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```

Optimization

- Lets see how well this works on the test set...



```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad! (SOTA is ~99.7%)

Optimization

- Strategy #2: **Follow the slope**

Optimization

- Strategy #2: **Follow the slope**

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient The direction of steepest descent is the **negative gradient**

Optimization

current W:

$[0.34,$
$-1.11,$
$0.78,$
$0.12,$
$0.55,$
$2.81,$
$-3.1,$
$-1.5,$
$0.33, \cdots]$

loss 1.25347

gradient dW:

$[?,$
$?,$
$?,$
$?,$
$?,$
$?,$
$?,$
$?,$
$?, \cdots$
$]$

Optimization

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33, …]
loss 1.25347

W + h (first dim)**:**

[0.34 +
0.0001,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33, …]
loss 1.25322

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?, …
]

Optimization

current W:

[0.34,
−1.11,
0.78,
0.12,
0.55,
2.81,
−3.1,
−1.5,
0.33, …]

loss 1.25347

W + h (first dim):

[0.34 + **0.0001**,
−1.11,
0.78,
0.12,
0.55,
2.81,
−3.1,
−1.5,
0.33, …]

loss 1.25322

gradient dW:

[**-2.5**,
?,

(1.25322 − 1.25347)/0.0001
= −2.5

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

?,
?,
…]

Optimization

current W:

[0. 34,
−1. 11,
0. 78,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, …]
loss 1.25347

W + h (second dim)**:**

[0. 34,
−1. 11 +
0.0001,
0. 78,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, …]
loss 1.25353

gradient dW:

[−
2. 5,
?,
?,
?,
?,
?,
?,
?,
?, …
]

Optimization

current W:

[0. 34,
−1. 11,
0. 78,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, …]
loss 1.25347

W + h (second dim)**:**

[0. 34,
−1. 11 + **0.0001**,
0. 78,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, …]
loss 1.25353

gradient dW:

[−2. 5,
0.6,
?,

(1. 25353 − 1. 25347) /0. 0001 = 0. 6

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

?, …
]

Optimization

current W:

[0. 34,
−1. 11,
0. 78,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, •••]

loss 1.25347

W + h (third dim)**:**

[0. 34,
−1. 11,
0. 78 + **0.0001**,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, •••]

loss 1.25347

gradient dW:

[−2. 5,
0. 6,
?,
?,
?,
?,
?,
?,
?, •••
]

Optimization

current W:

[0.34,
−1.11,
0.78,
0.12,
0.55,
2.81,
−3.1,
−1.5,
0.33, …]
loss 1.25347

W + h (third dim)**:**

[0.34,
−1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
−3.1,
−1.5,
0.33, …]
loss 1.25347

gradient dW:

[−2.5,
0.6,
0,
?,
…]

(1.25347 − 1.25347)/0.0001 = 0

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Optimization

current W:

[0. 34,
−1. 11,
0. 78,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, …]

loss 1.25347

W + h (third dim)**:**

[0. 34,
−1. 11,
0. 78 + **0.0001**,
0. 12,
0. 55,
2. 81,
−3. 1,
−1. 5,
0. 33, …]

loss 1.25347

gradient dW:

[−2. 5,
0. 6,
0,
?,
…]

Numeric Gradient

- Slow! Need to loop over all dimensions
- Approximate

Optimization

This is silly. The loss is just a function of W:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

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

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Optimization

This is silly The loss is just a function of W:

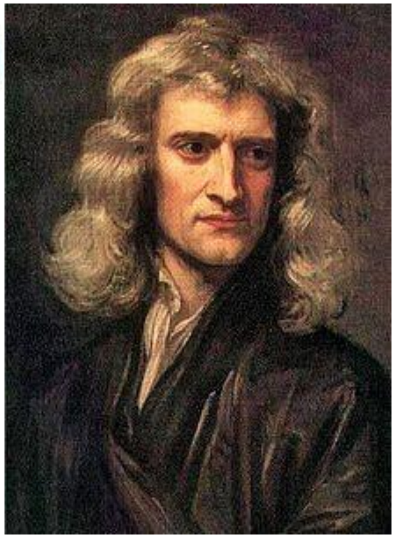
$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

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

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Use calculus to compute an **analytic gradient**


This image is in the public domain

This image is in the public domain

Optimization

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33, …]

loss 1.25347

dW = ...
(some function data and W)

gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,
…]

Optimization

In summary:

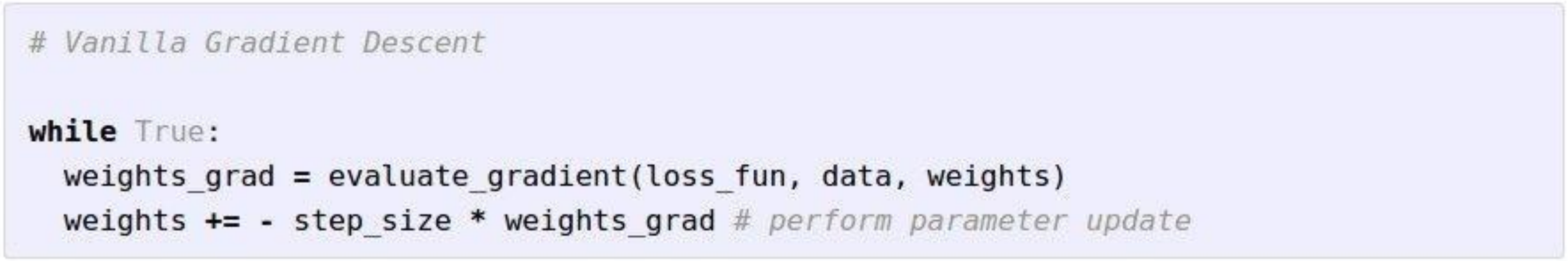
- Numerical gradient: approximate, slow easy to write
- Analytic gradient: exact, fast, error-prone

=>

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

Optimization

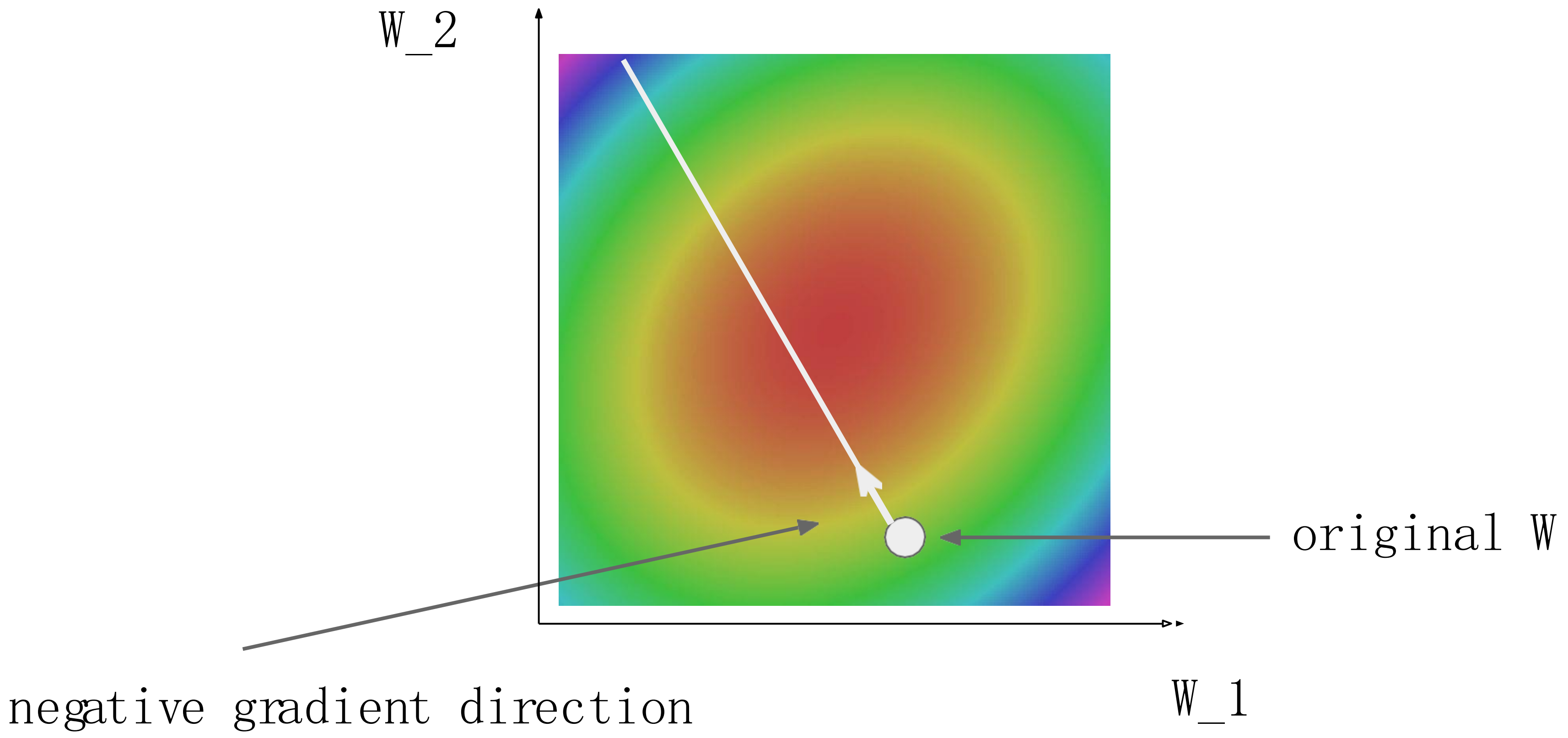
- Gradient Descent



```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Optimization



Stochastic Gradient Descent(SGD)

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

Full sum expensive when N is large!

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

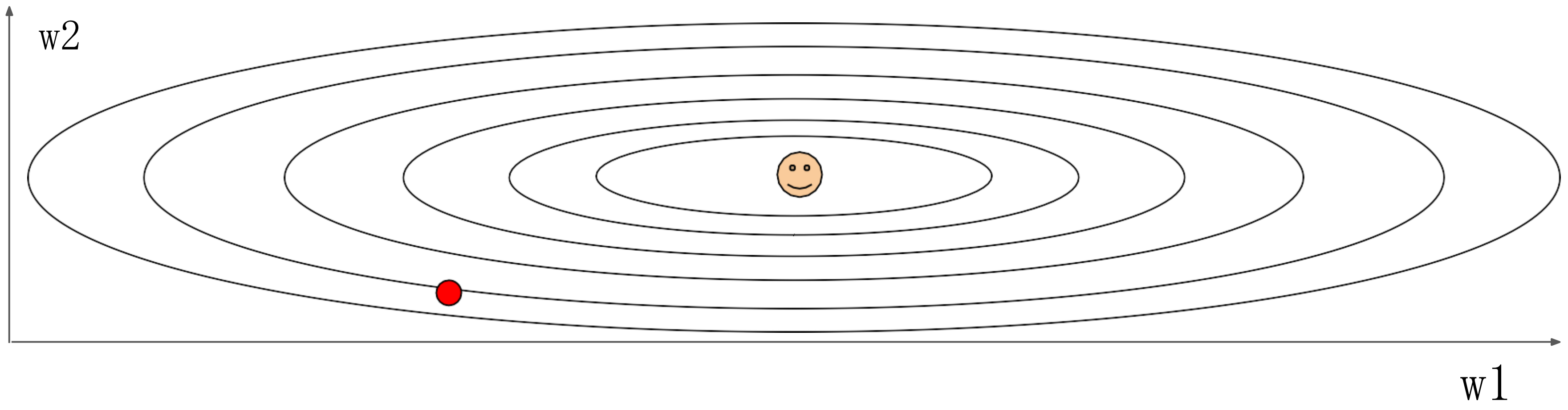
Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Optimization: Problem #1 with SGD

What if loss changes quickly in one direction and slowly in another? What does gradient descent do?

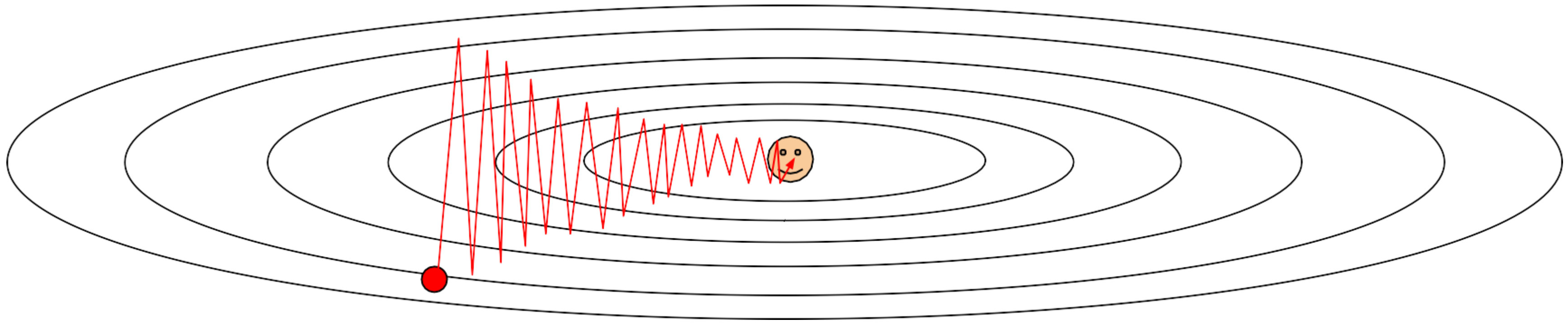


Aside: Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large.

Optimization: Problem #1 with SGD

What if loss changes quickly in one direction and slowly in another? What does gradient descent do?

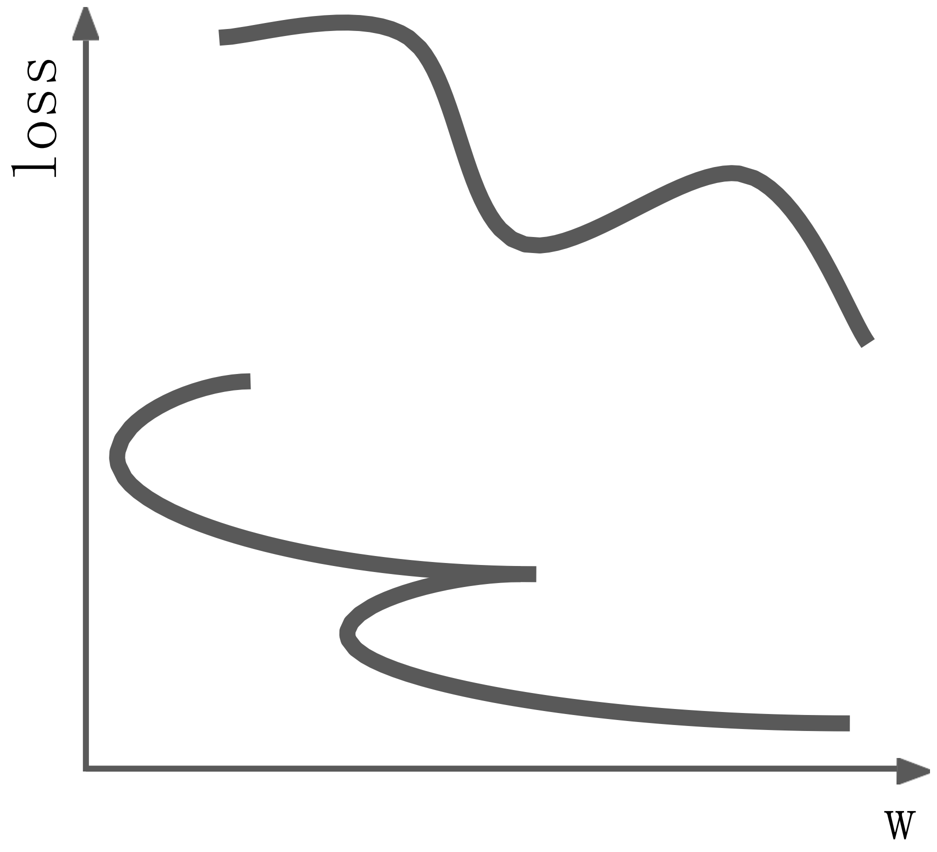
Very slow progress along shallow dimension, jitter along steep direction



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

Optimization: Problem #2 with SGD

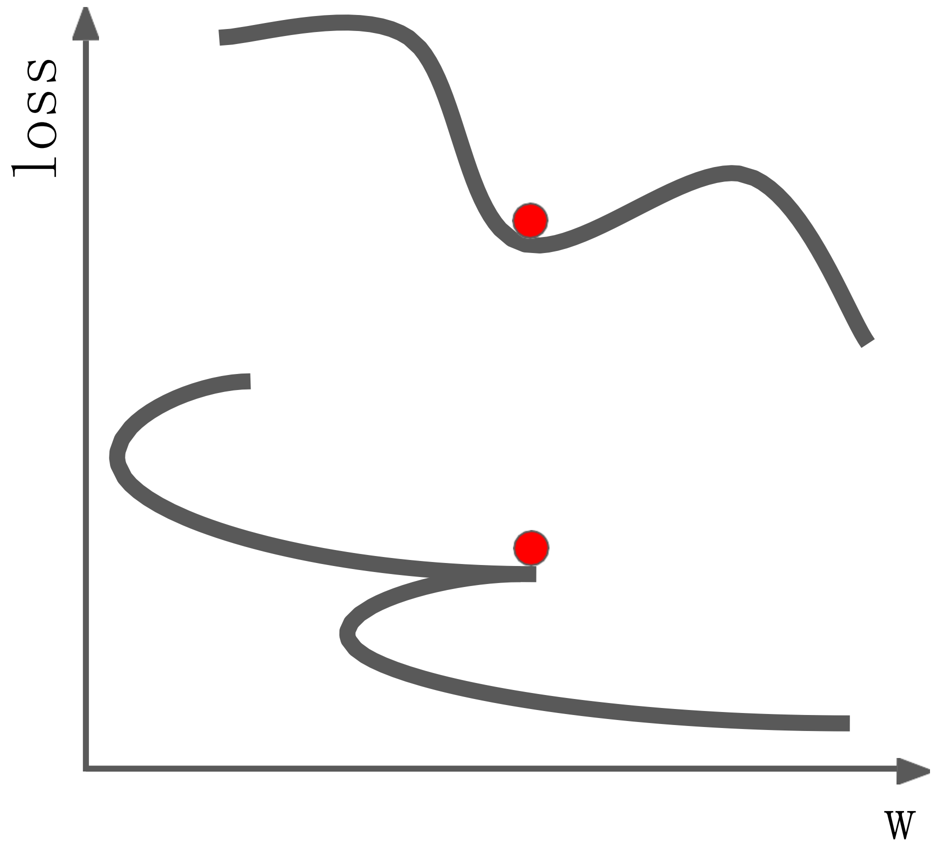
What if the loss function has a **local minima** or **saddle point**?



Optimization: Problem #2 with SGD

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

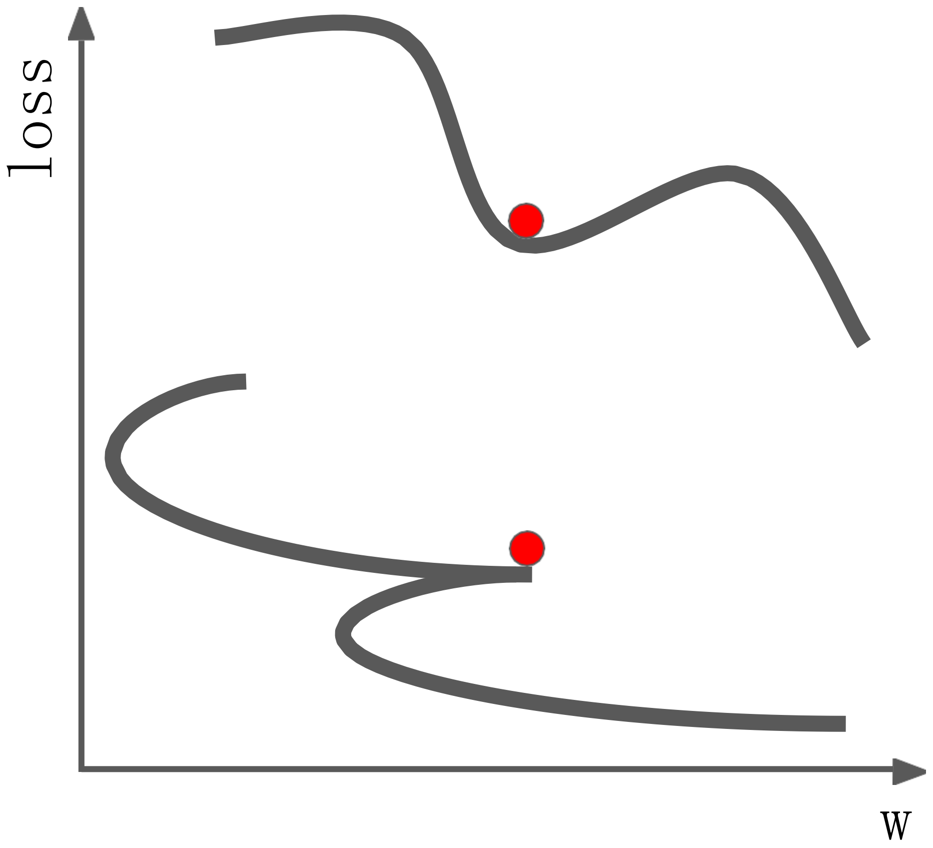
Zero gradient, gradient descent gets stuck



Optimization: Problem #2 with SGD

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

Saddle points much more common in high dimension



Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

Optimization: Problem #2 with SGD

saddle point in two dimension

$$f(x, y) = x^2 - y^2$$

$$\frac{\partial}{\partial x}(x^2 - y^2) = 2x \to 2(0) = 0$$

$$\frac{\partial}{\partial y}(x^2 - y^2) = -2y \to -2(0) = 0$$

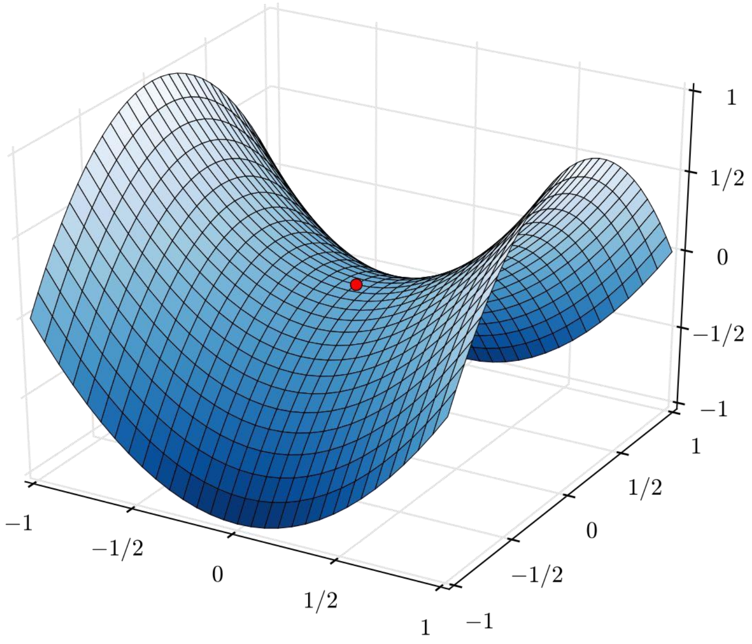


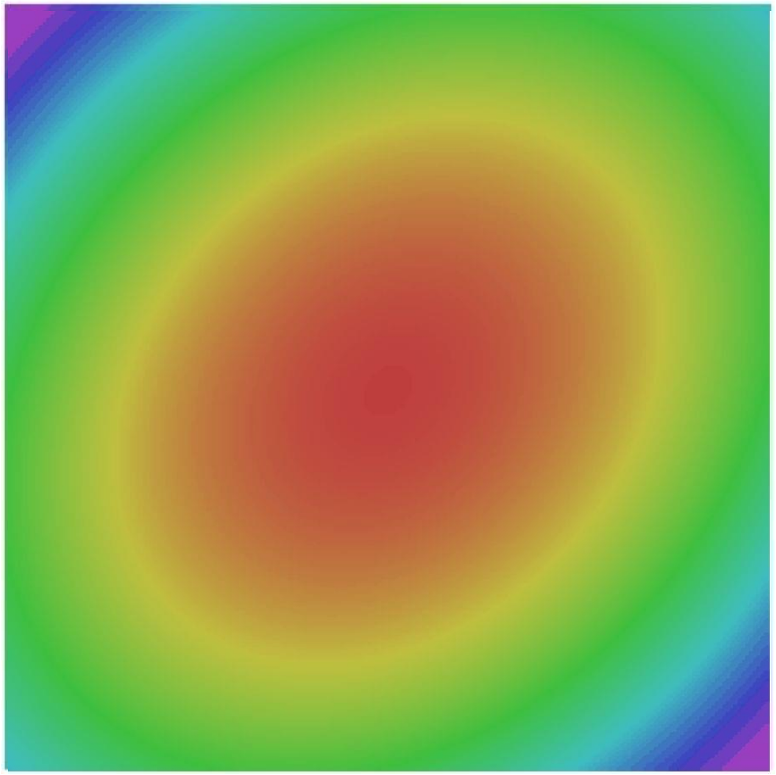
Image source:
https://en.wikipedia.org/wiki/Saddle_point

Optimization: Problem #3 with SGD

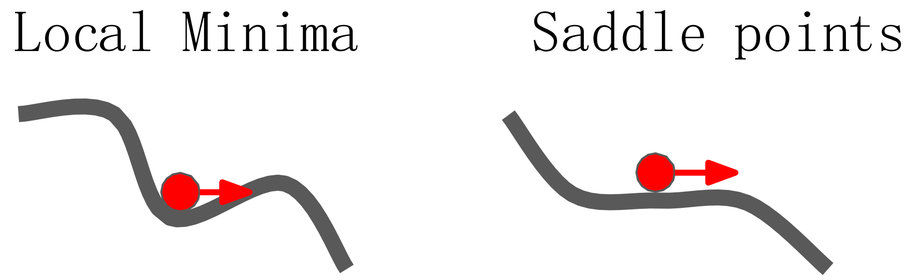
Our gradients come from minibatches so they can be noisy!

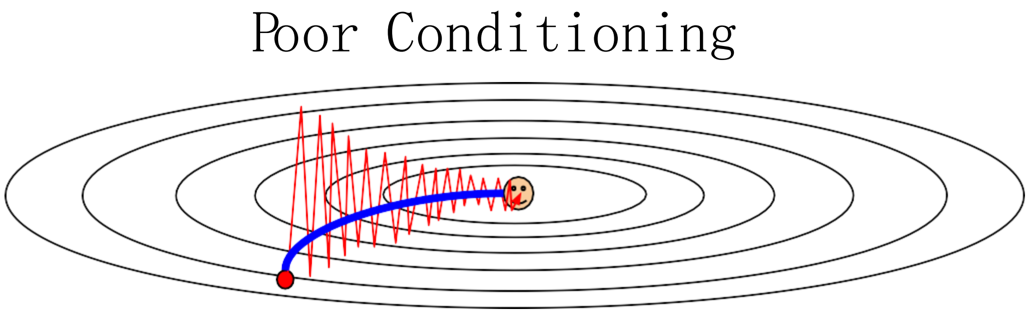
$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

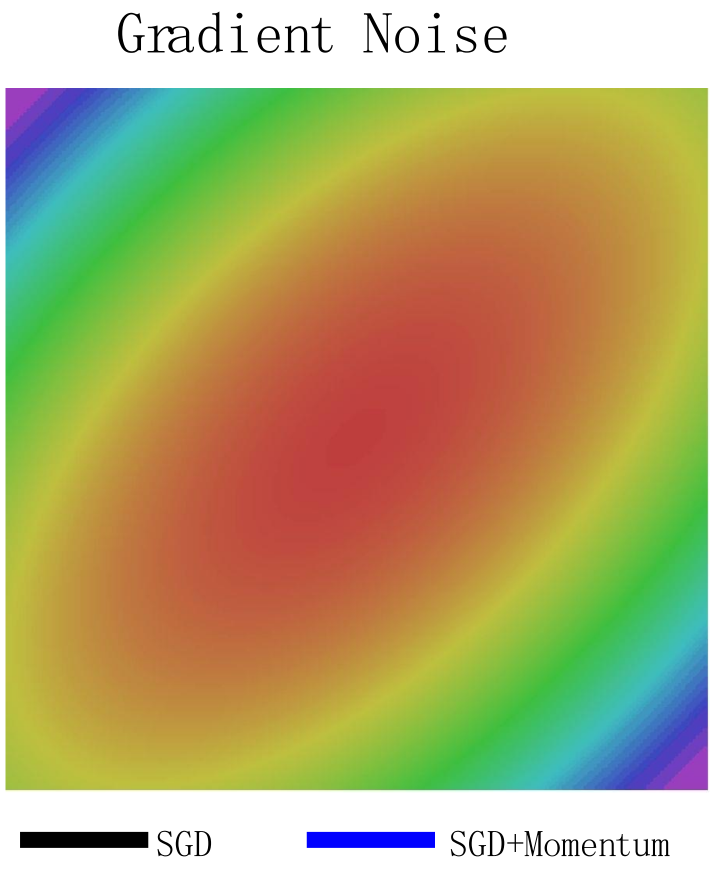
$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



SGD + Momentum

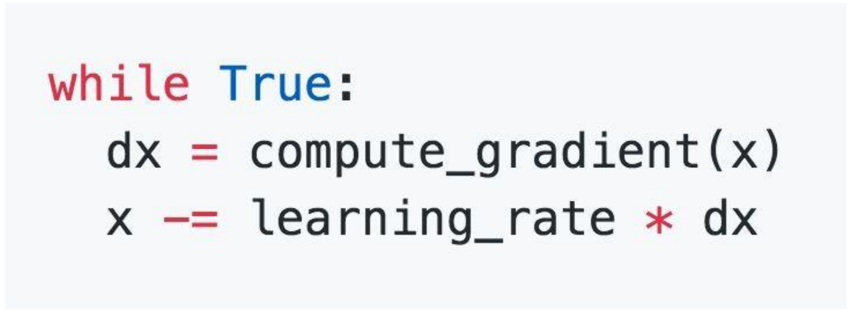






SGD: the simple two line update code

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



```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD + Momentum

continue moving in the general direction as the previous iterations

SGD

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

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

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

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- 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

alternative equivalent formulation

SGD+Momentum

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

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx - learning_rate * dx
  x += vx
```

SGD+Momentum

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

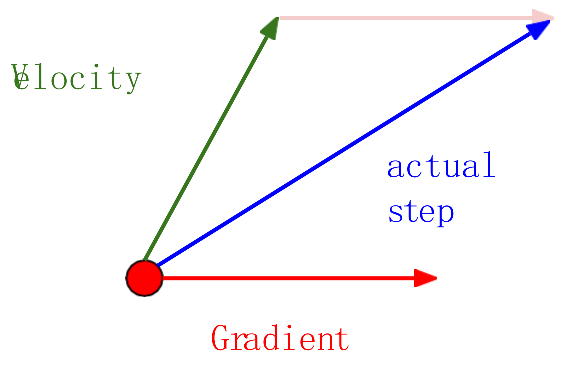
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

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

Momentum update:

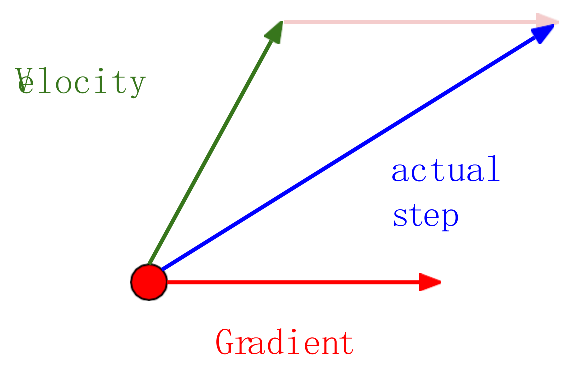


Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983 Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Nesterov Momentum

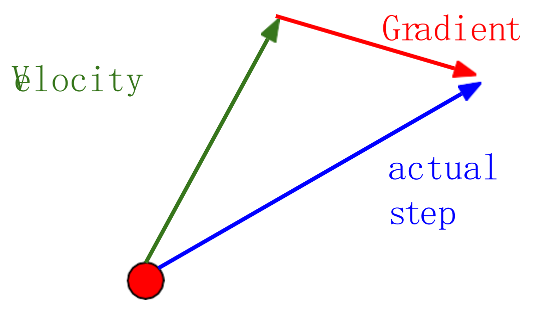
Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983 Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov Momentum

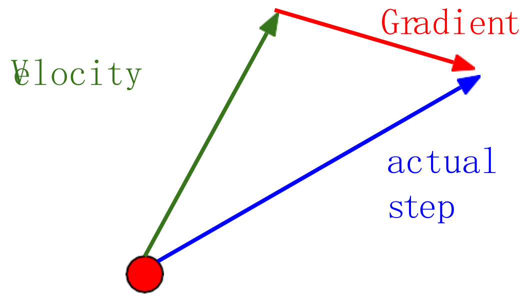


"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

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

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$



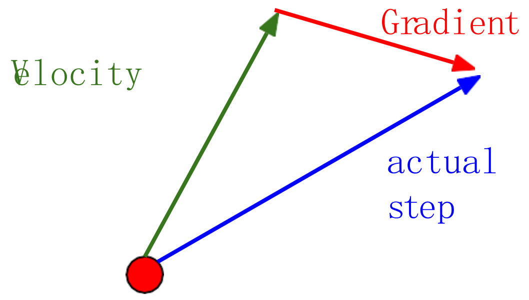
“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

$$\boxed{\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t}) \\ x_{t+1} &= x_t + v_{t+1} \end{aligned}}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = \boxed{\boxed{x_t} + \rho v_t}$ and rearrange:



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

https://cs231n.github.io/neural-networks-3/

Nesterov Momentum

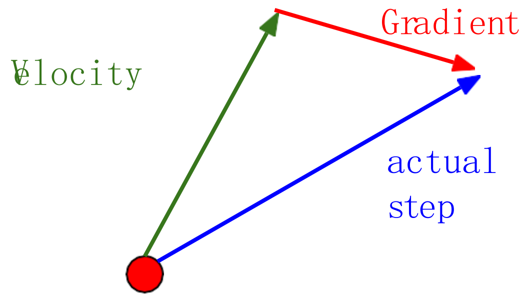
$$\boxed{\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t}) \\ x_{t+1} &= x_t + v_{t+1} \end{aligned}}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

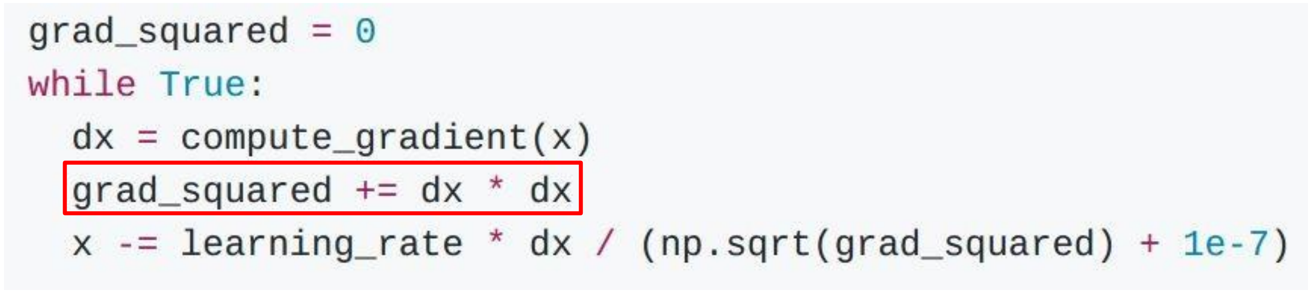
$$\boxed{\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\ \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \end{aligned}}$$

https://cs231n.github.io/neural-networks-3/



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

AdaGrad



```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

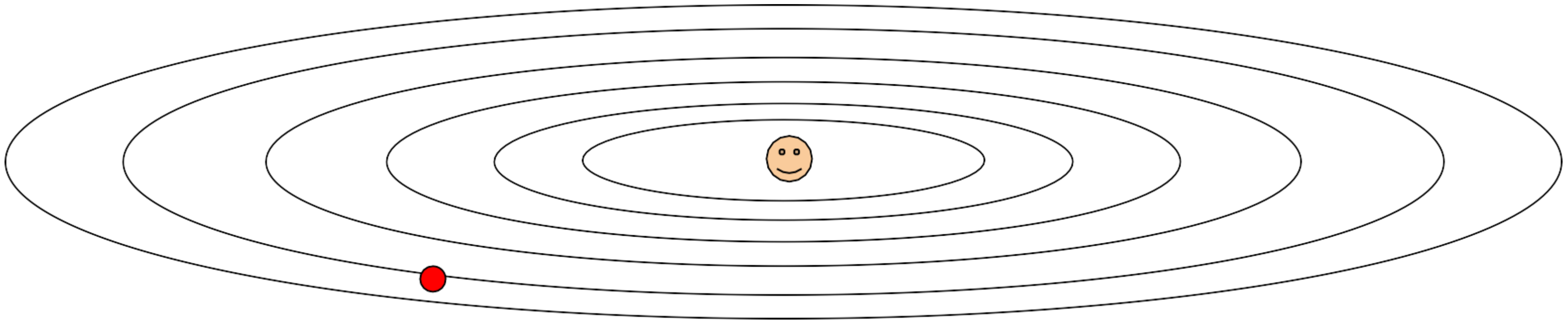
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates" or
"adaptive learning rates"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

AdaGrad

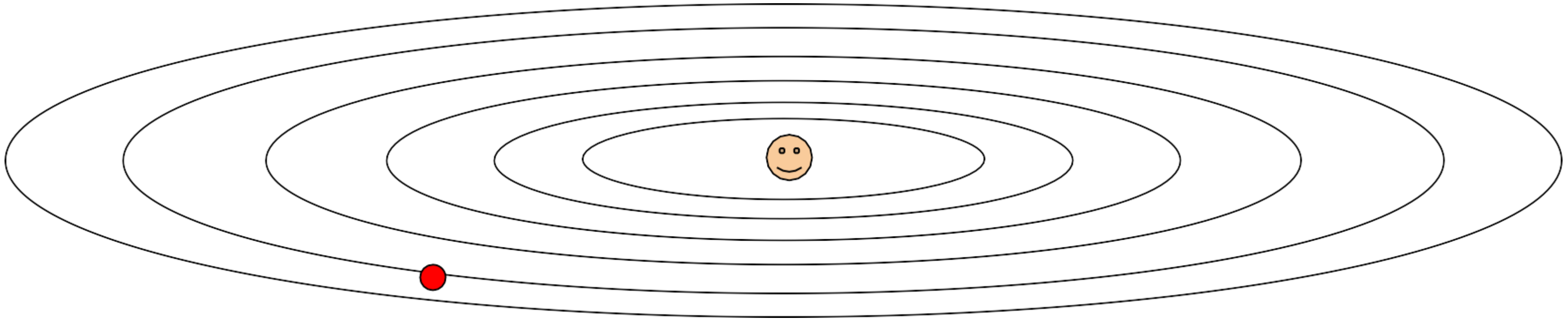
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

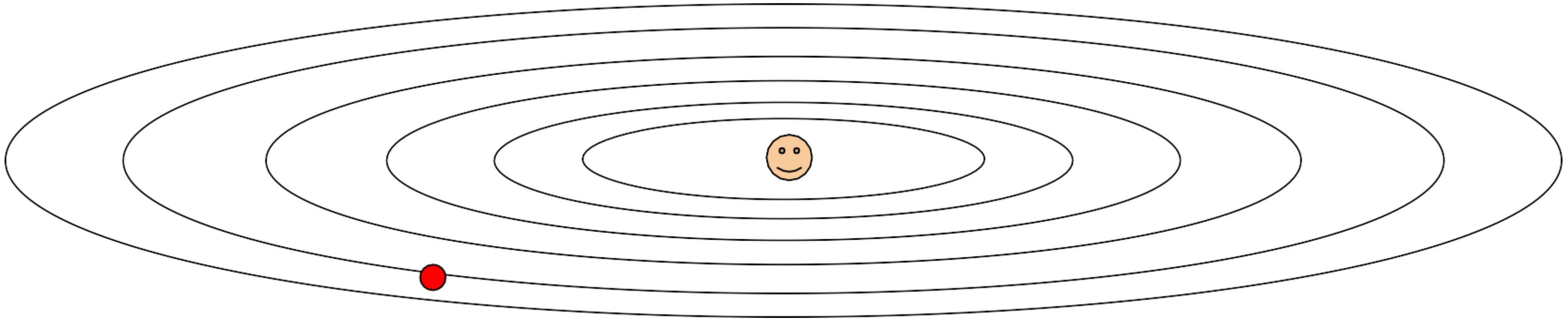


Q: What happens with AdaGrad?

Progress along "steep" directions is damped; progress along "flat" directions is accelerated

AdaGrad

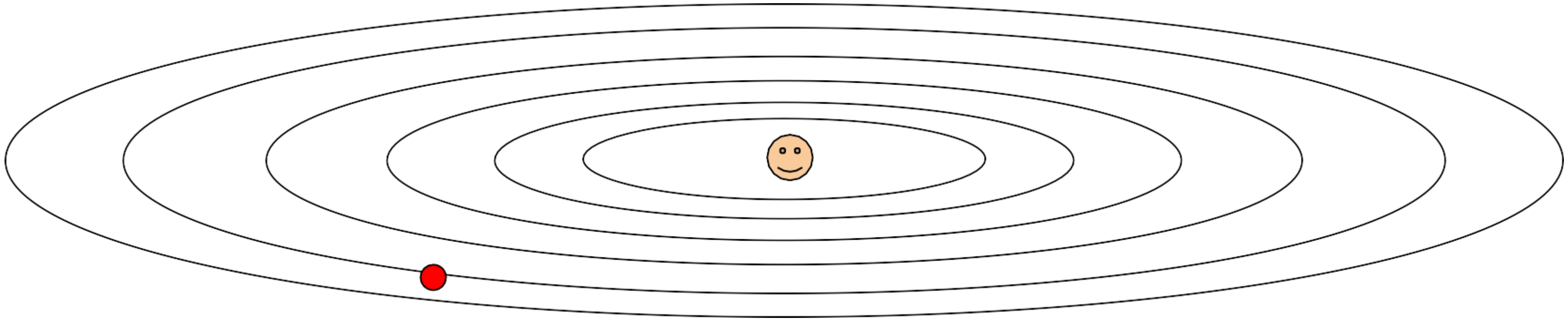
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q2: What happens to the step size over long time?

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

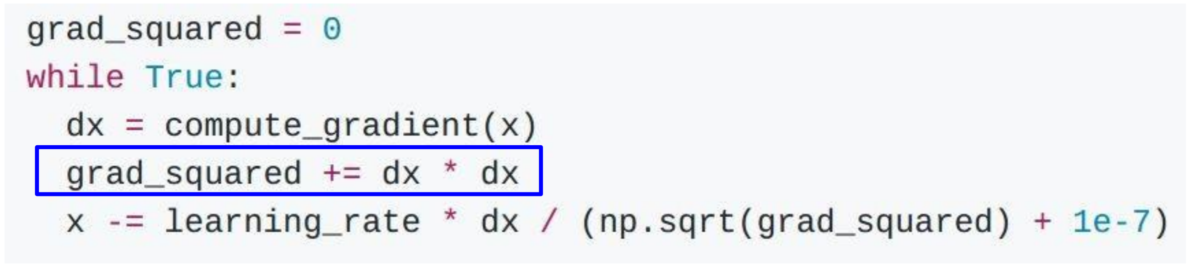


Q2: What happens to the step size over long time?

Decays to zero

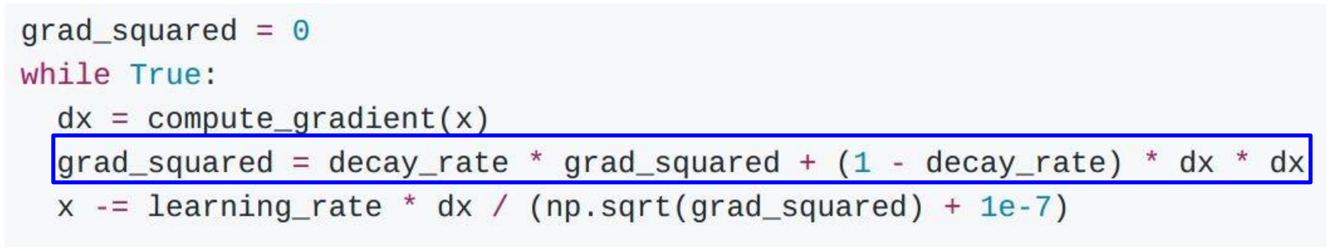
RMSProp: “Leaky AdaGrad”

AdaGrad



```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

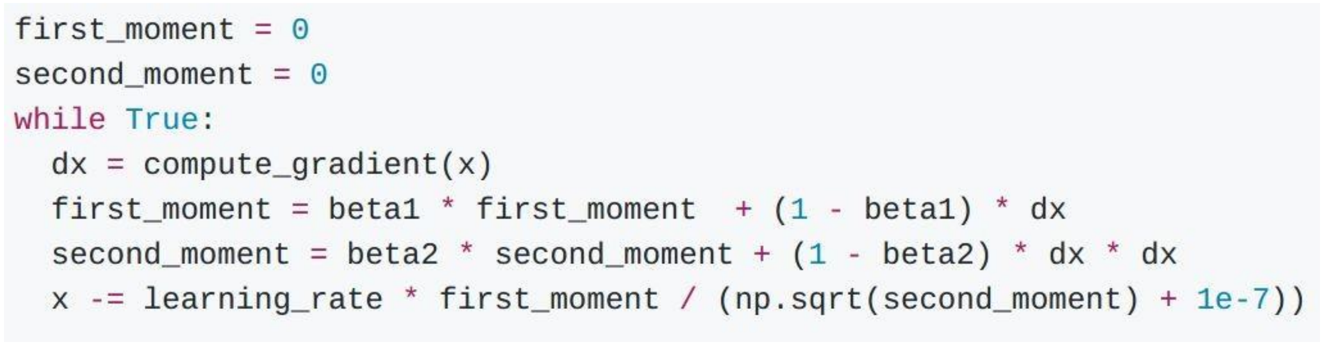
RMSProp



```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Adam (almost)



```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

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

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

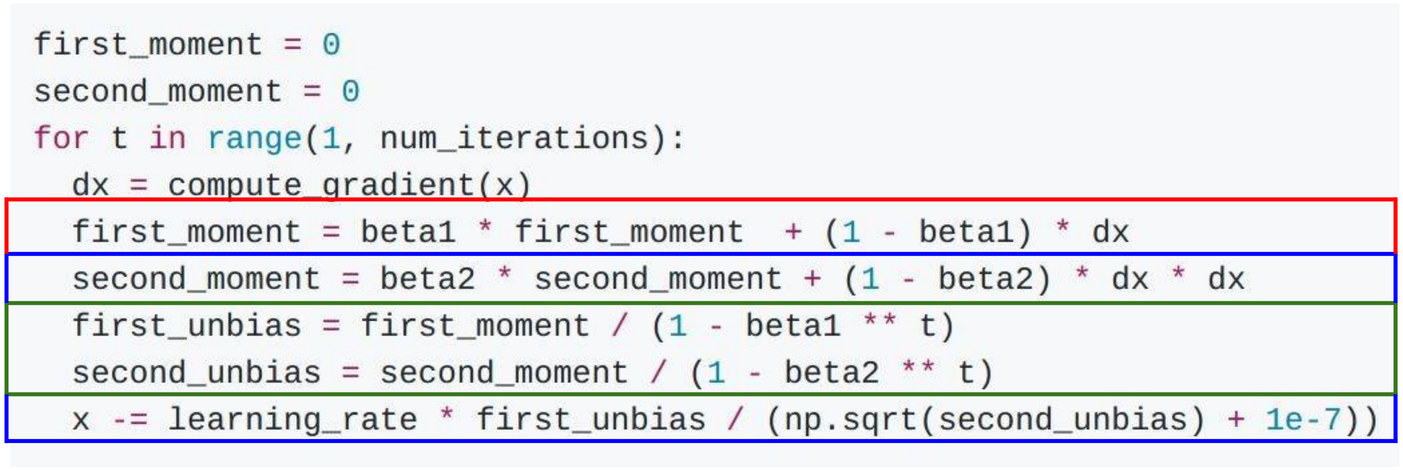
AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

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

Adam (full form)



```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

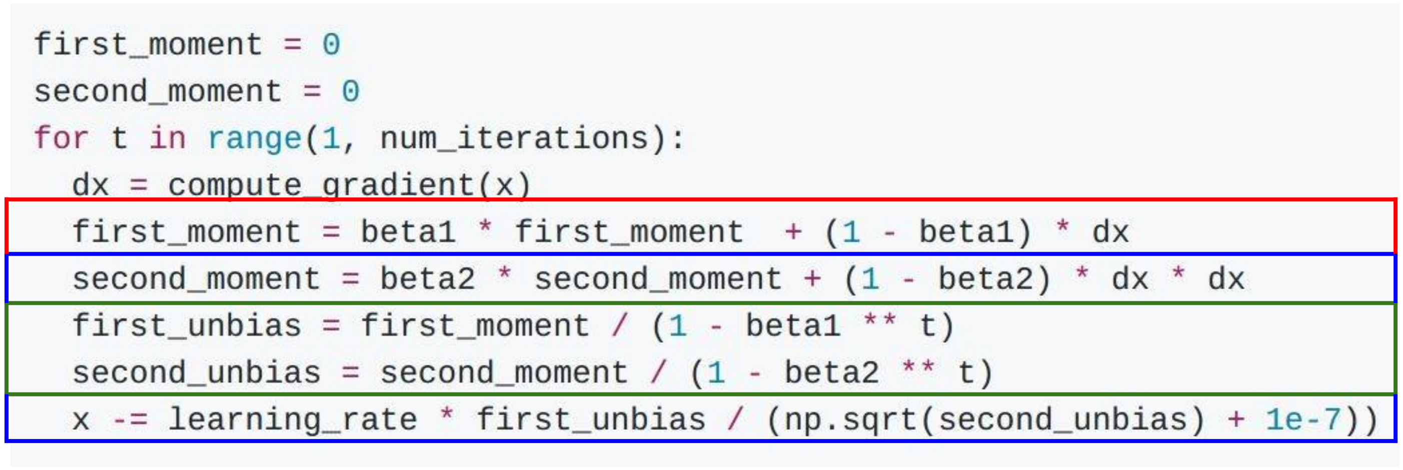
Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

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

Adam (full form)



```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

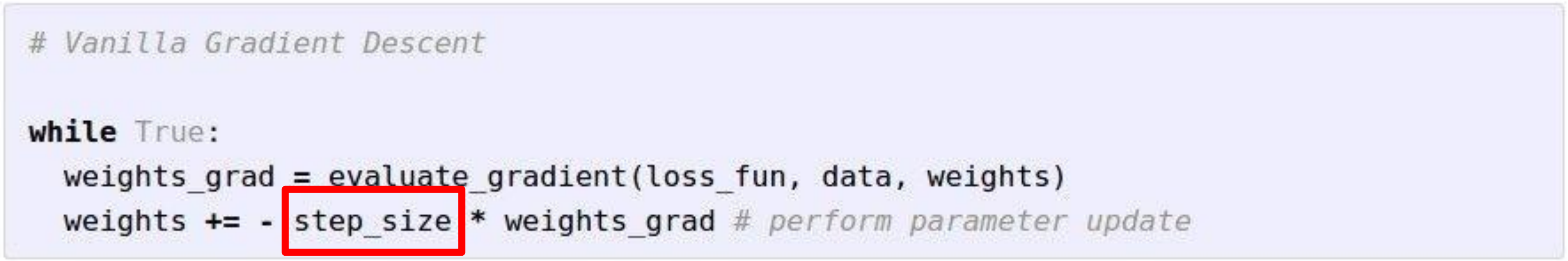
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

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

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

Learning rate schedules



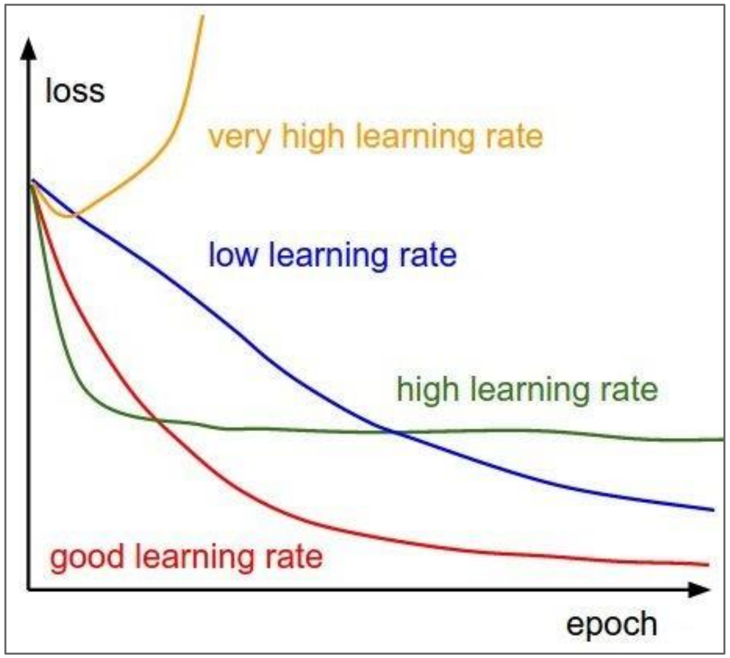
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Learning rate

Learning rate schedules

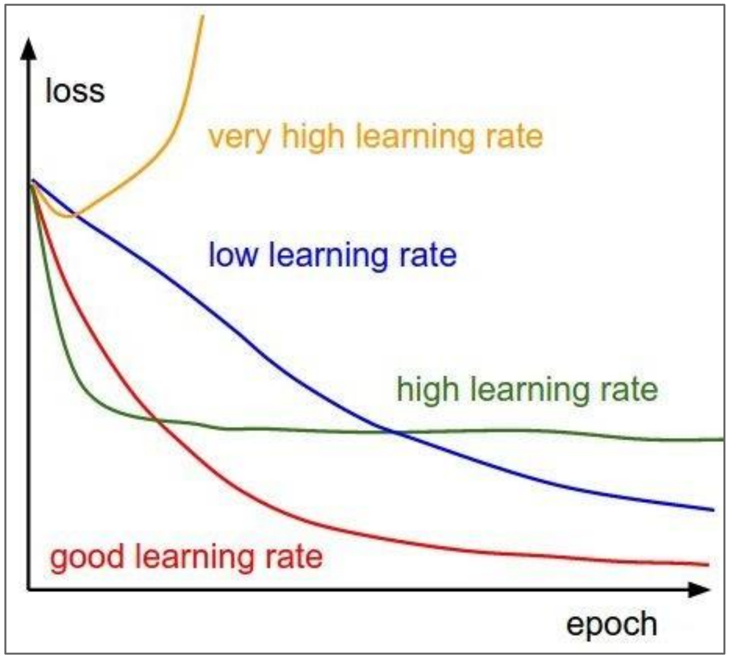
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter



Q: Which one of these learning rates is best to use?

Learning rate schedules

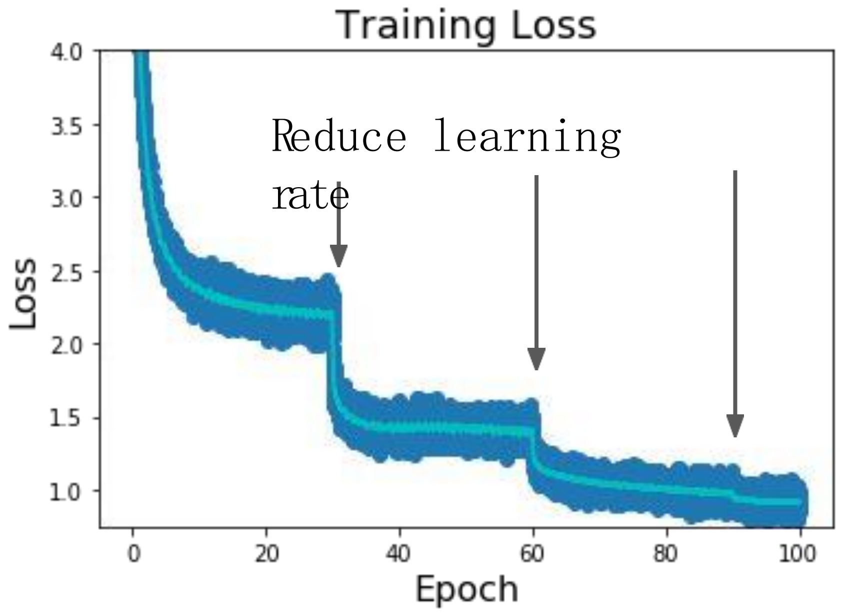
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter



Q: Which one of these learning rates is best to use?

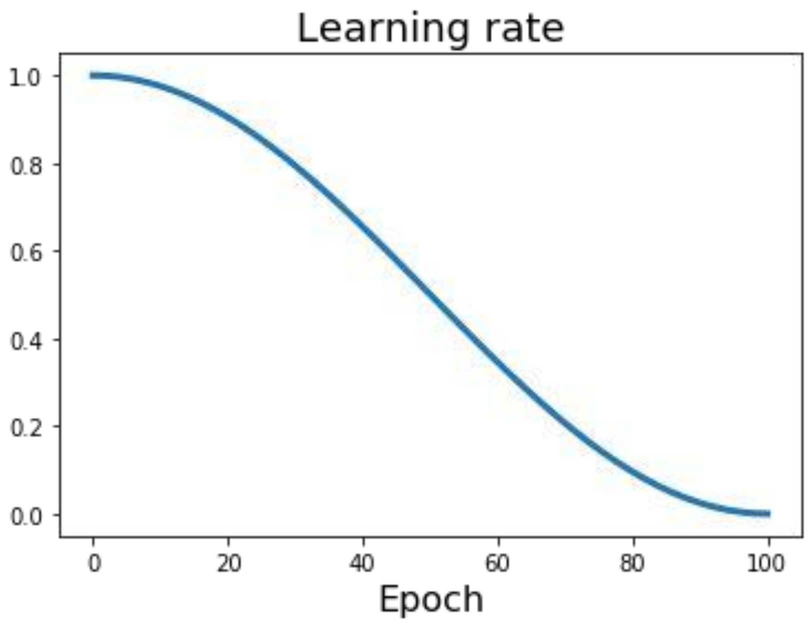
A: In reality all of these are good learning rates.

Learning rate decays over time



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Learning Rate Decay



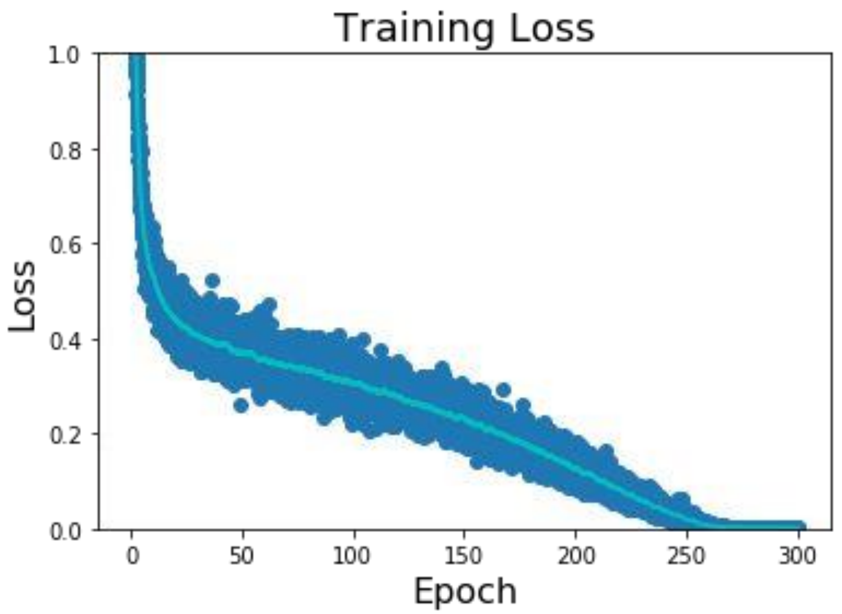
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017 Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018 Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

Learning Rate Decay



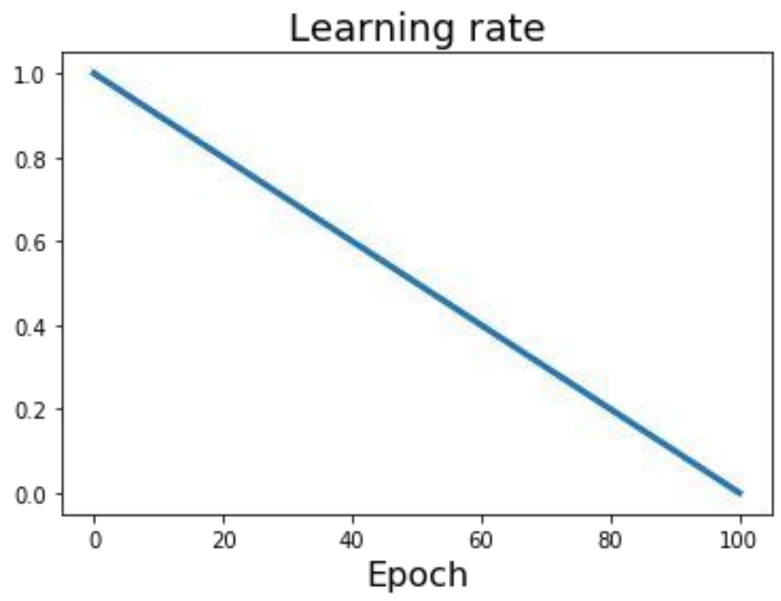
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017 Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018 Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

Learning Rate Decay



Vaswani et al, “Attention is all you need”, NIPS 2017

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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

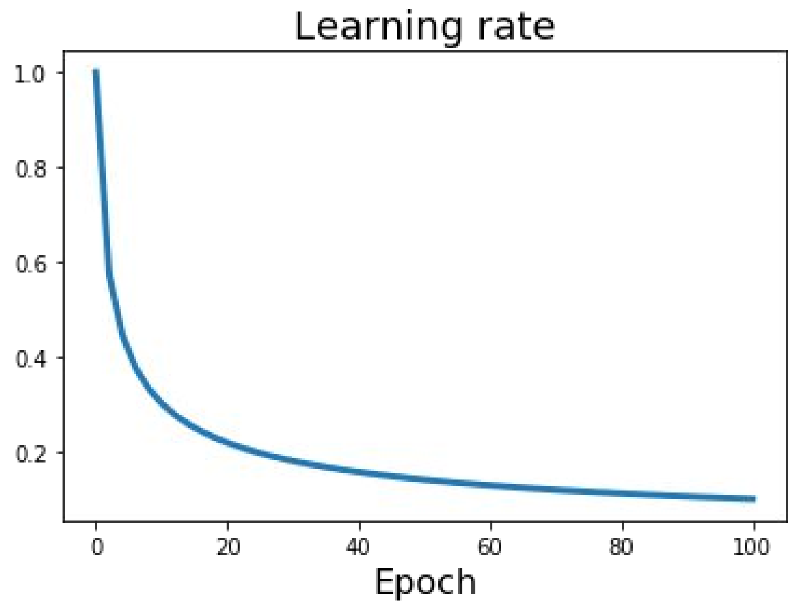
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Linear: $\alpha_t = \alpha_0(1 - t/T)$

Learning Rate Decay



Vaswani et al, “Attention is all you need”, NIPS 2017

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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

α_0 : Initial learning rate

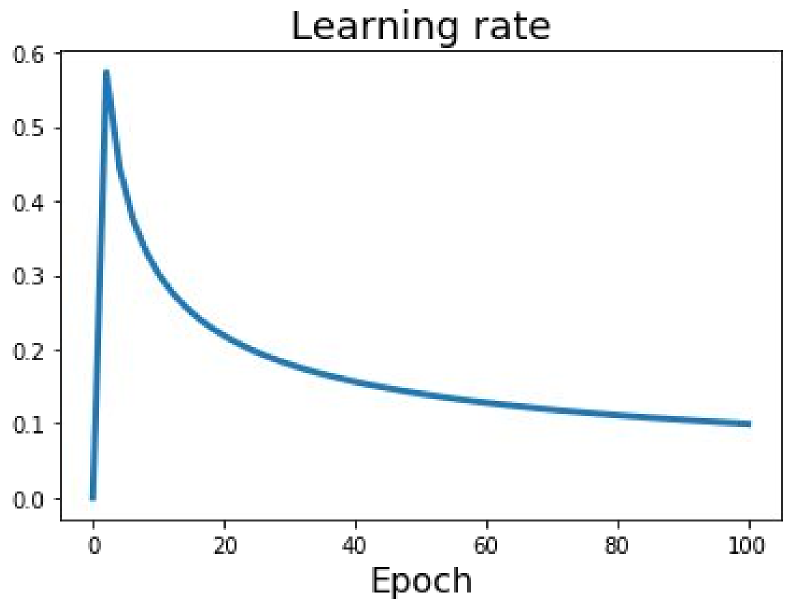
α_t : Learning rate at epoch t

T : Total number of epochs

Linear: $\alpha_t = \alpha_0(1-t/T)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

Learning Rate Decay: Linear Warmup

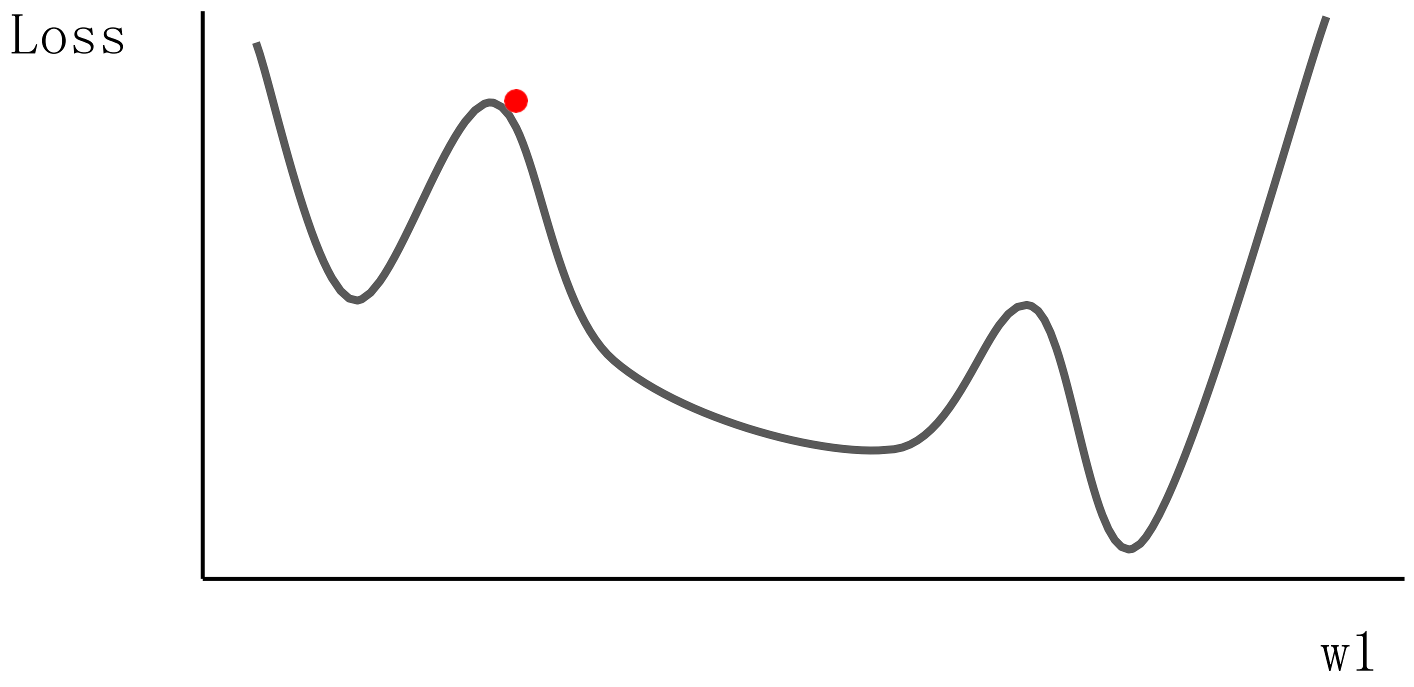


High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5,000 iterations can prevent this.

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N

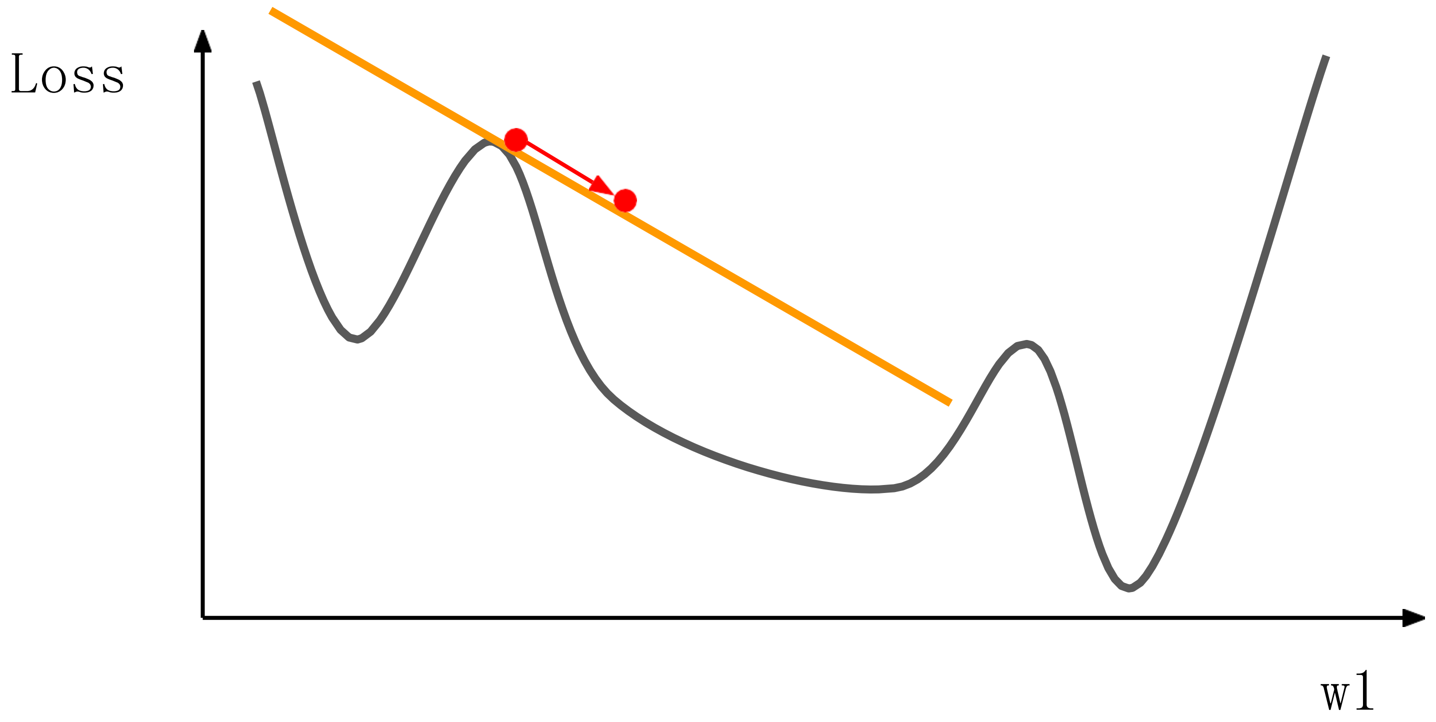
Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

First-Order Optimization



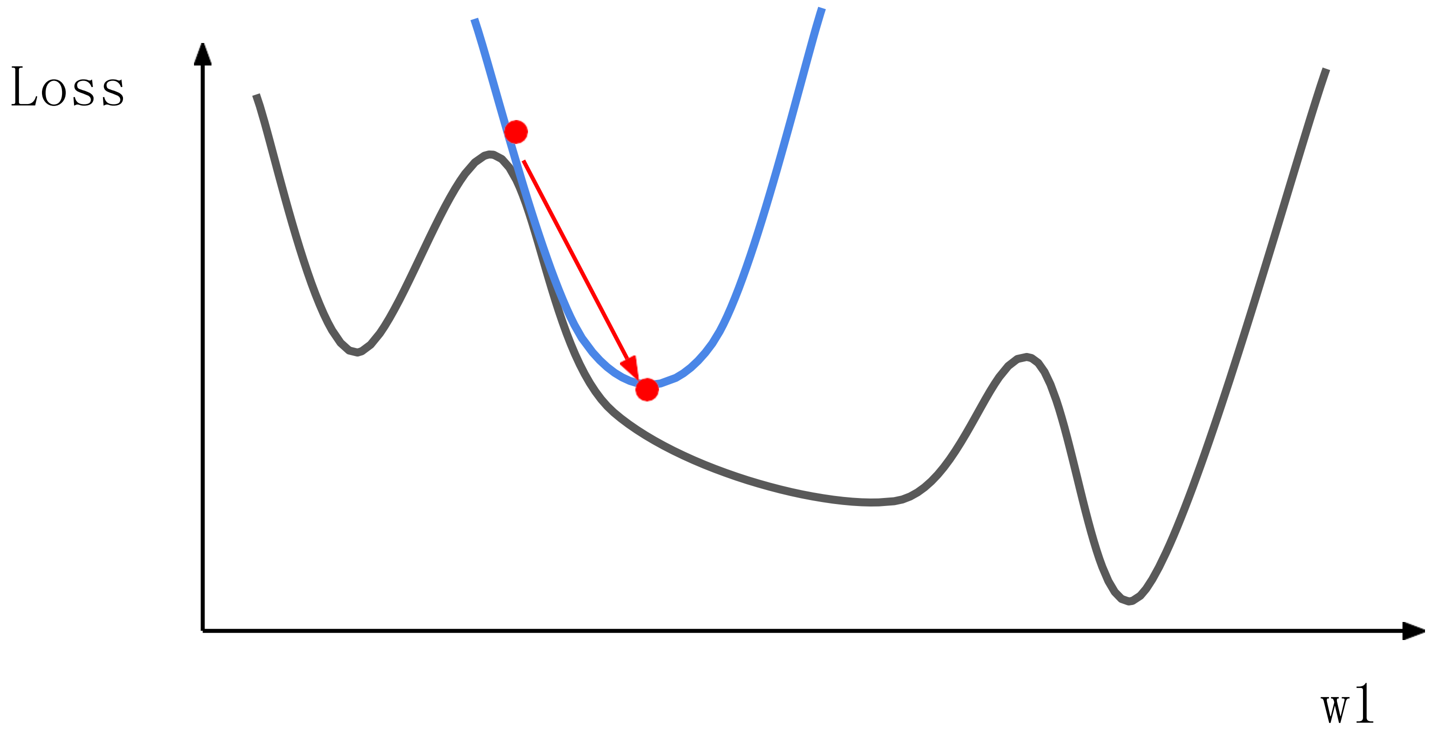
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation



Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q: Why is this bad for deep learning?

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of Millions)

Q: Why is this bad for deep learning?

Second-Order Optimization

$$\boxed{\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)}$$

- Quasi-Newton methods (**BGFS** most popular): *instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).*

- **L-BFGS** (Limited memory BFGS): *Does not form/store the full inverse Hessian.*

L-BFGS

- **Usually works very well in full batch, deterministic mode**
  i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**.
  Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Le et al, “On optimization methods for deep learning, ICML 2011”
Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017

In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate

- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)

中山大學
SUN YAT-SEN UNIVERSITY

Next time:

Neural Networks and Backpropagation

Pattern Recognition and Computer Vision

Guanbin Li,
School of Computer Science and Engineering, Sun Yat-Sen University