ROB 430/599: Deep Learning for Robot Perception and Manipulation (DeepRob)

Lecture 4: Regularization and Optimization

01/21/2026

Today

- Announcement, Feedback, and Recap (5min)
- Regularization and Optimization
  - Regularization (15min)
  - Optimization (20min)
  - Computing Gradients (30min)
- Summary and Takeaways (5min)

Project Tips

- Upgrade to Google Colab Pro
  - "session crash"/low RAM issue: Try using a high-RAM runtime
- Submission on Autograder
  - upload ALL files, even if some are empty (not worked on yet).

Grading

● Programming Projects (individual)	**(52%)**
○ Project 0	5%
○ Project 1	11%
○ Project 2	12%
○ Project 3	12%
○ Project 4	12%
● Midterm (individual)	**(10%)**
● Final Project (Group)	**(23%)**
○ Proposal Presentation	5%
○ Final Report and code (paper reproduction, algorithmic extension)	15%
○ Showcase (Video, Website, etc.)	3%
● In-class activities (individual) [quiz, notebooks, etc.]	**(10%)**
● **Participation (individual)** [in class participation, office hours, Piazza, etc.]	**(5%)**

Total: 100%

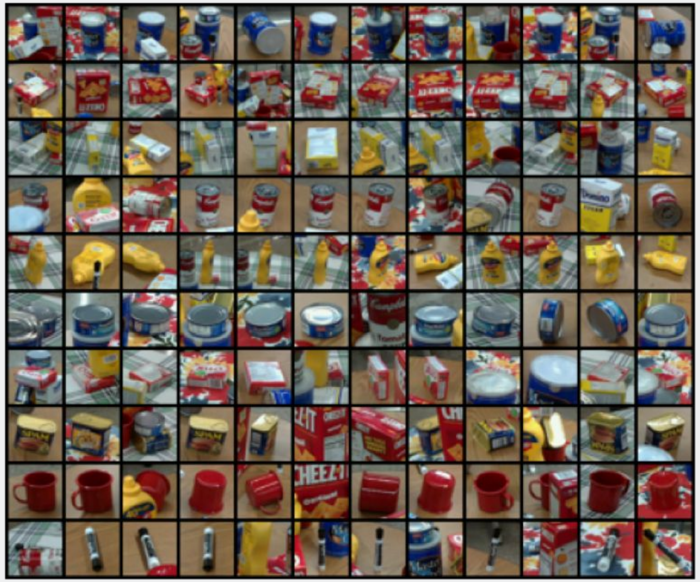
Grading

- Participation: Ahaslides
  - Lectures + subset of discussions
  - Officially beginning week of Jan 26 (next Monday)
  - Five (5) absences, no questions asked
  - Late arrival due to class schedule conflict: no penalty
  - Attending OH / Post on Piazza: not required to receive full participation credits
- Late policy for project submission
  - Three (3) late tokens, each 24 hrs
  - After late tokens are used: late penalties (10% per day)
- Details in course information doc

Project 1 - Dataset

Deadline: Feb. 1, 2026

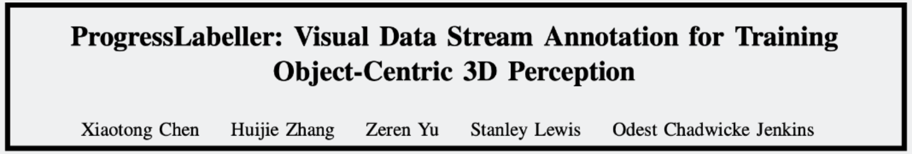
Progress **R**obot **O**bject **P**erception **S**amples **D**ataset



Chen et al., "ProgressLabeller: Visual Data Stream Annotation for Training Object-Centric 3D Perception", IROS, 2022.

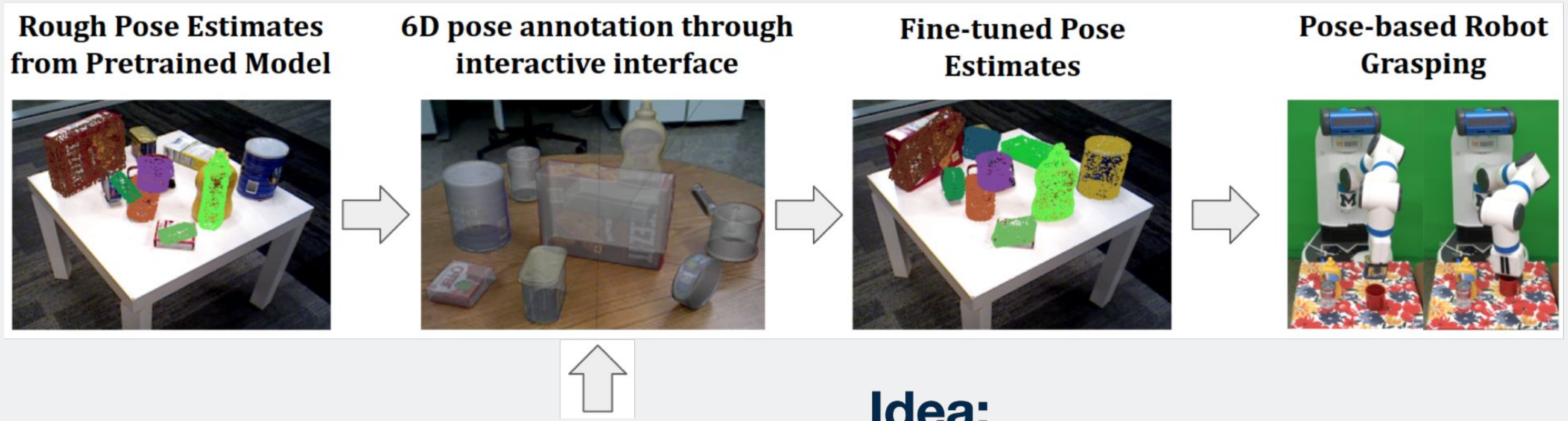
10 classes
32x32 RGB images
50k training images (5k per class)
10k test images (1k per class)

Project 1 - How was this dataset created?



ProgressLabeller: Visual Data Stream Annotation for Training Object-Centric 3D Perception

Xiaotong Chen Huijie Zhang Zeren Yu Stanley Lewis Odest Chadwicke Jenkins


Rough Pose Estimates from Pretrained Model
6D pose annotation through interactive interface
Fine-tuned Pose Estimates
Pose-based Robot Grasping

Human Annotator

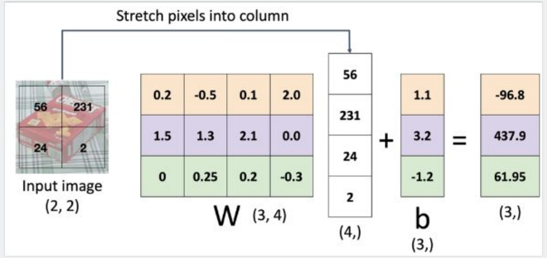
Idea:

1. **Record video of scene**
2. **Human labels object pose in selected frames**
3. **Pose labels propagate to (large number of) remaining frames**

Recap: Linear Classifier - Three Viewpoints

① **Algebraic Viewpoint**

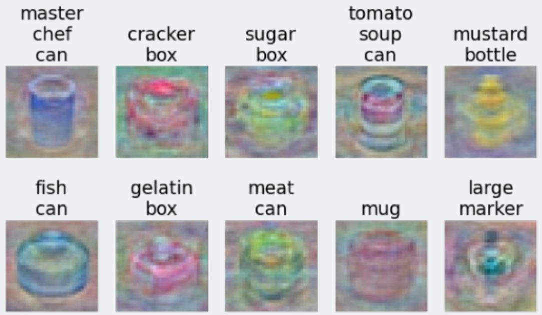
$f(x,W) = Wx$



$$\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix} + \begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix} = \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}$$

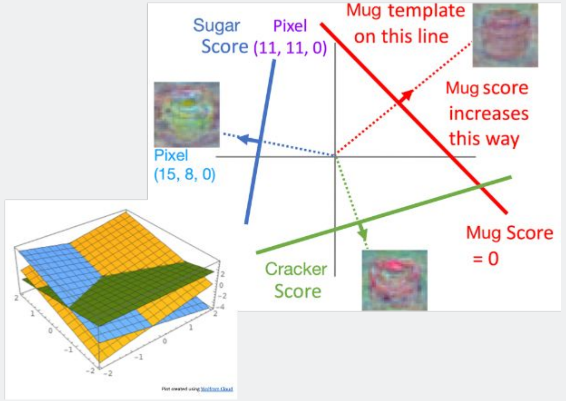
② **Visual Viewpoint**

One template per class



③ **Geometric Viewpoint**

Hyperplanes cutting up space



Recap: Loss Functions

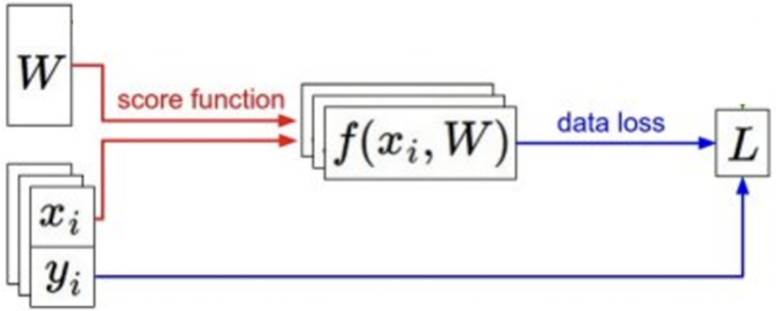
- We have some dataset of (x, y)
- We have a **score function:**
- We have a **loss function**:

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

Linear classifier

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

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



Discussion on Last week’s Quizzes

(refer to Canvas)

- If you have questions, please come ask!

How to find the best W and b?

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

Linear classifier

Problem: Loss functions encourage good performance on training data but we care about test data (i.e., generalization)

Regularization

Overfitting

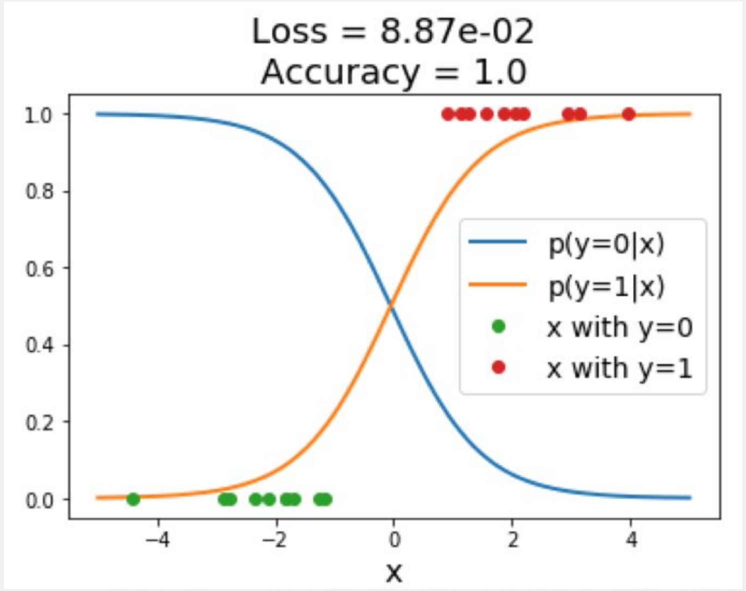
A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

Example: Linear classifier with 1D inputs, 2 classes, and softmax loss

$$s_i = w_i x + b_i$$

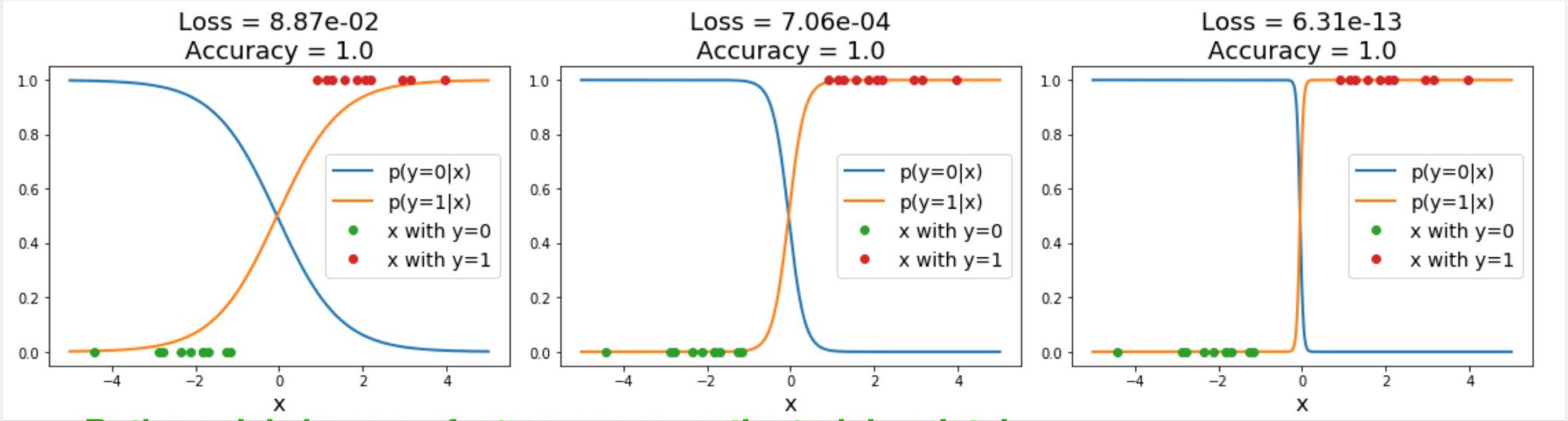
$$p_i = \frac{exp(s_i)}{exp(s_1) + exp(s_2)}$$

$$L = -log(p_y)$$



Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data



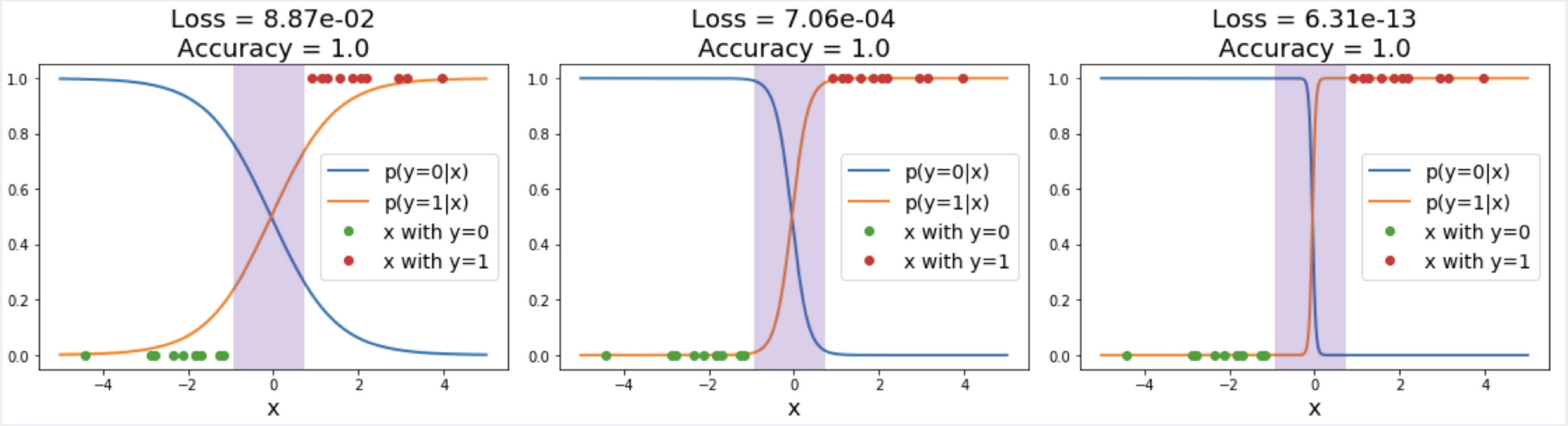
Both models have perfect accuracy on the training data!

Low loss, but unnatural "cliff" between the training points

ROBOTICS

Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data



Overconfidence in regions with no training data could give **poor generalization**

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}}$$

Data loss: Model predictions should match training data

ROBOTICS

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \underbrace{\boxed{\lambda R(W)}}_{\text{Regularization: Prevent the model from doing too well on training data}}$$

Data loss: Model predictions should match training data

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

ROBOTICS

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \underbrace{\boxed{\lambda} R(W)}_{\text{Regularization: Prevent the model from doing too well on training data}}$$

λ → Hyperparameter giving regularization strength

Data loss: Model predictions should match training data

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

Simple examples:

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

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

More complex:
Dropout
Batch normalization
Cutout, Mixup, Stochastic depth, etc…

Regularization: Prefer Simpler Models

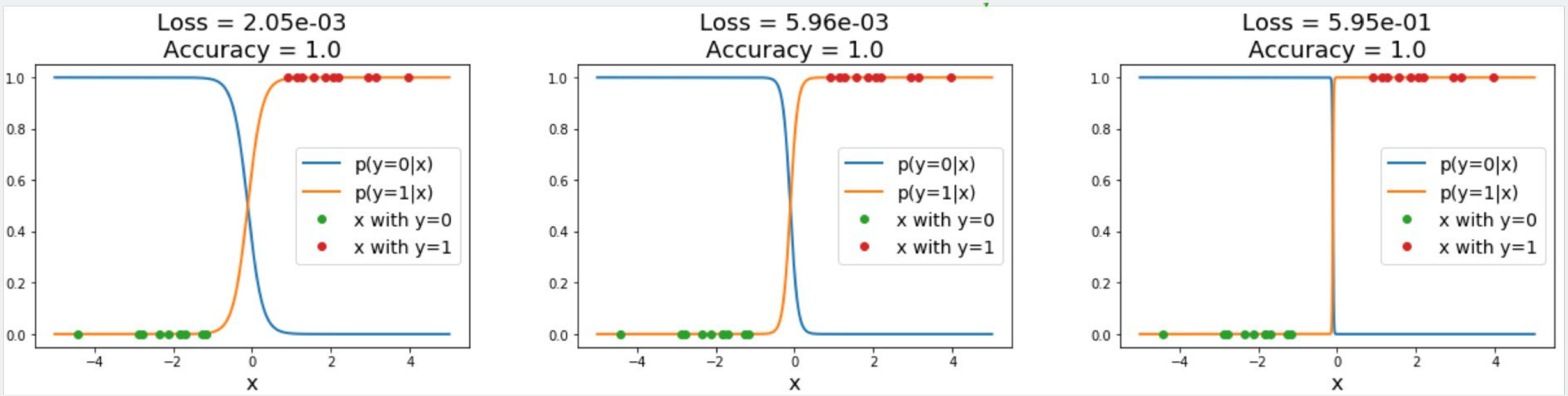
Example: Linear classifier with 1D inputs, 2 classes, and softmax loss

$$s_i = w_i x + b_i$$

$$p_i = \frac{exp(s_i)}{exp(s_1) + exp(s_2)}$$

$$L = -log(p_y) + \lambda \sum_i w_i^2$$

Regularization term causes loss to **increase** for model with sharp cliff



Aha Slides
(In-class participation)

https://ahaslides.com/J5D83

Regularization: Expressing Preferences

$$x = [1,1,1,1]$$
$$w_1 = [1,0,0,0]$$
$$w_2 = [0.25,0.25,0.25,0.25]$$

L2 Regularization

$$\boxed{R(W) = \sum_{k,l} W_{k,l}^2}$$

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

Q1: Which weight would the data loss prefer?
Q2: Which weight would the L2 regularization prefer?

Hint: what does it mean by "prefer"? Higher? Lower?

Optimization

Finding a good W

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

Loss function consists of **data loss** to fit the training data and **regularization** to prevent overfitting

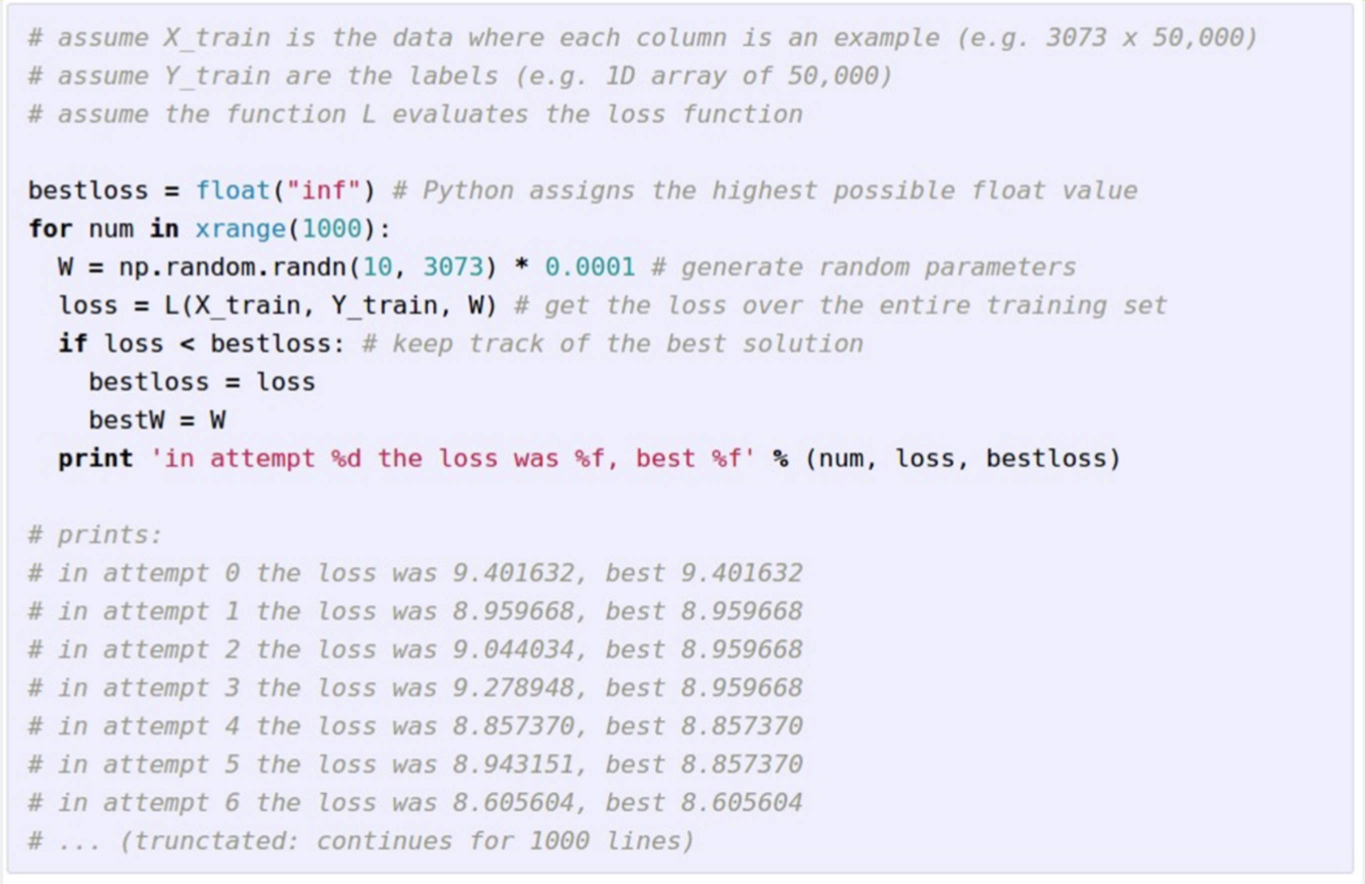
Optimization

$$\boxed{w^* = \arg\min_{w} L(w)}$$

Optimization

$$w^* = \arg\min_{w} L(w)$$

Idea 1: Random Search (bad idea!)

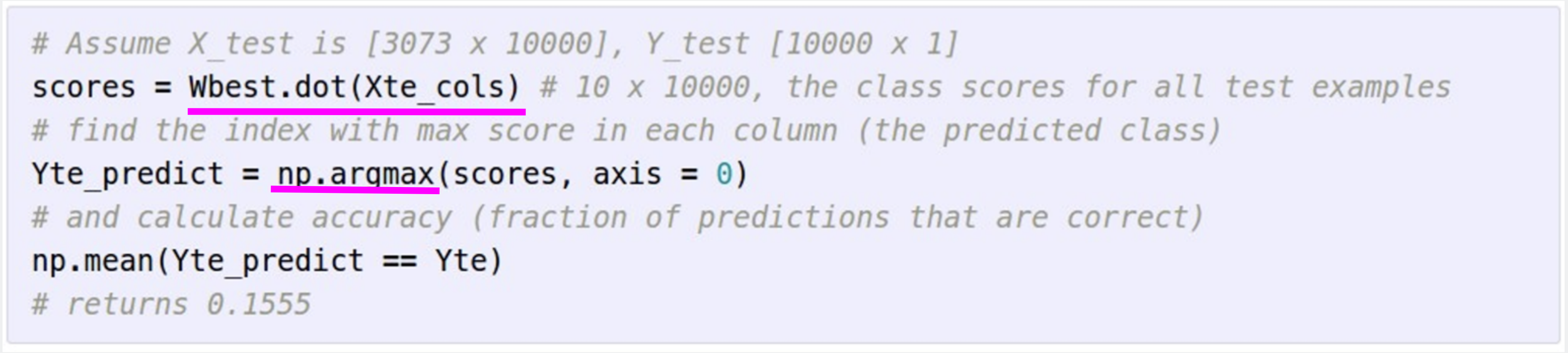


```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

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)
```

Idea 1: Random Search (bad idea!)



```
# 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 on CIFAR-10

Idea 2: Follow the slope

$$\boxed{w^* = \arg\min_{w} L(w)}$$

The valley image and the walking man image are in CC0 1.0 public domain

Idea 2: Follow the slope

$$\boxed{w^* = \arg\min_w L(w)}$$

In 1-dimension, the **derivative** of a function gives the slope:

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

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**.

Idea 2: Follow the slope

$$w^* = \arg\min_{w} L(w)$$

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

(example)

Current W:		Gradient $\frac{dL}{dW}$
[0.34,		[?,
-1.11,		?,
0.78,		?,
0.12,		?,
0.55,		?,
2.81,		?,
-3.1,		?,
-1.5,		?,
0.33, …]		?, …]
loss 1.25347		

(example)

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

Current **W**:	**W + h** (first dim):	Gradient $\frac{dL}{dW}$
[0.34,	[0.34 + **0.0001**,	[?, ← ???
-1.11,	-1.11,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33, …]	0.33, …]	?, …]
loss 1.25347	loss 1.25322	

ROBOTICS

Aha Slides (In-class participation)

https://ahaslides.com/J5D83

Q3

(example)

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 $\frac{dL}{dW}$

0.6,
?,
?,

(1.25353 - 1.25347)/
0.0001
= 0.6

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

(example)

Current **W**:	**W + h** (third dim):	Gradient $\frac{dL}{dW}$
[0.34,	[0.34,	
-1.11,	-1.11,	0.6,
0.78,	0.78 + **0.0001**,	0.0,
0.12,	0.12	?
0.55		
2.81		
-3.1,		
-1.5,		
0.33, …]	0.33, …]	?, …]
loss 1.25347	loss 1.25347	

① **Numeric Gradient:**
- Slow: O(# dimensions)
- Approximate

ROBOTICS

Loss is 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

② Analytic gradient

(example)

Current **W**:

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

$\frac{dL}{dW}$ = some function of data and W

In practice we will compute $\frac{dL}{dW}$ using back propagation; see Lecture 6

Gradient $\frac{dL}{dW}$

[-2.5,
0.6,
0.0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1, …]

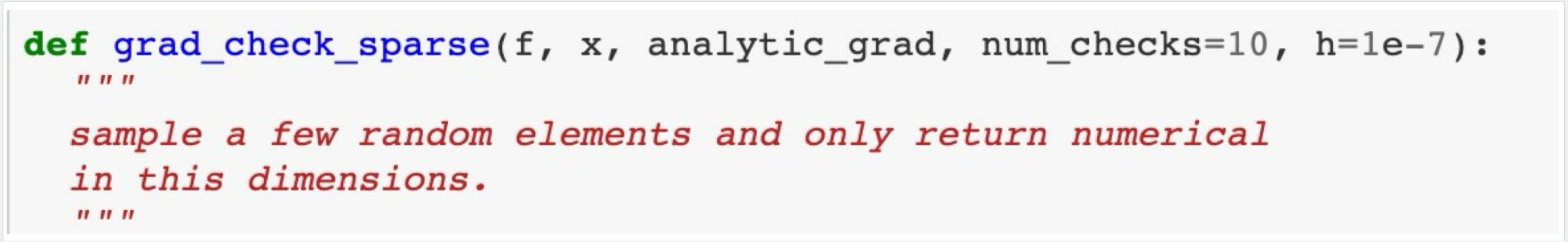
Computing Gradients

Computing Gradients

① **Numeric gradient:** approximate, slow, easy to write

② **Analytic gradient:** exact, fast, error-prone

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



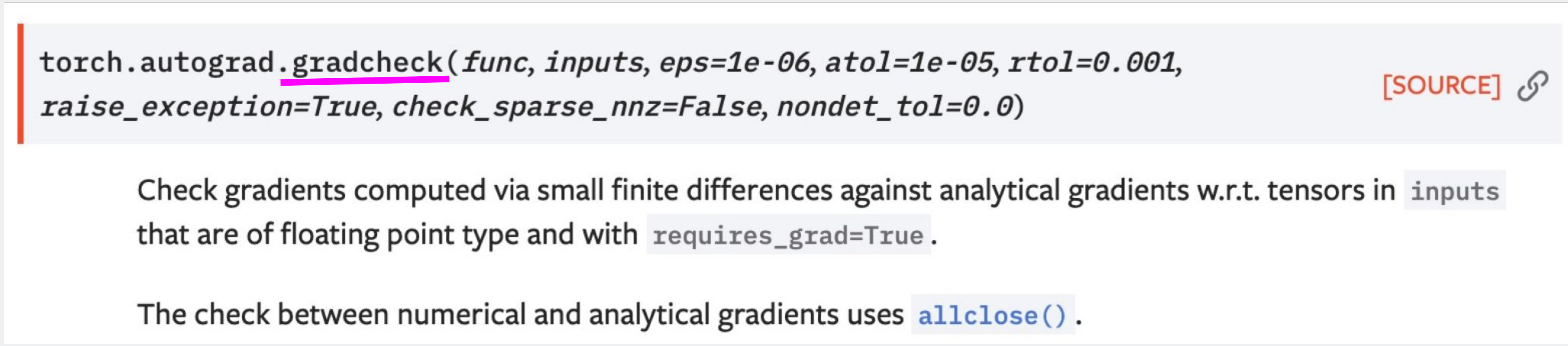
```
def grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-7):
  """
  sample a few random elements and only return numerical
  in this dimensions.
  """
```

Also check out: https://cs231n.github.io/optimization-1/
https://pytorch.org/docs/stable/notes/gradcheck.html

Computing Gradients

① **Numeric gradient:** approximate, slow, easy to write

② **Analytic gradient:** exact, fast, error-prone



```
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]

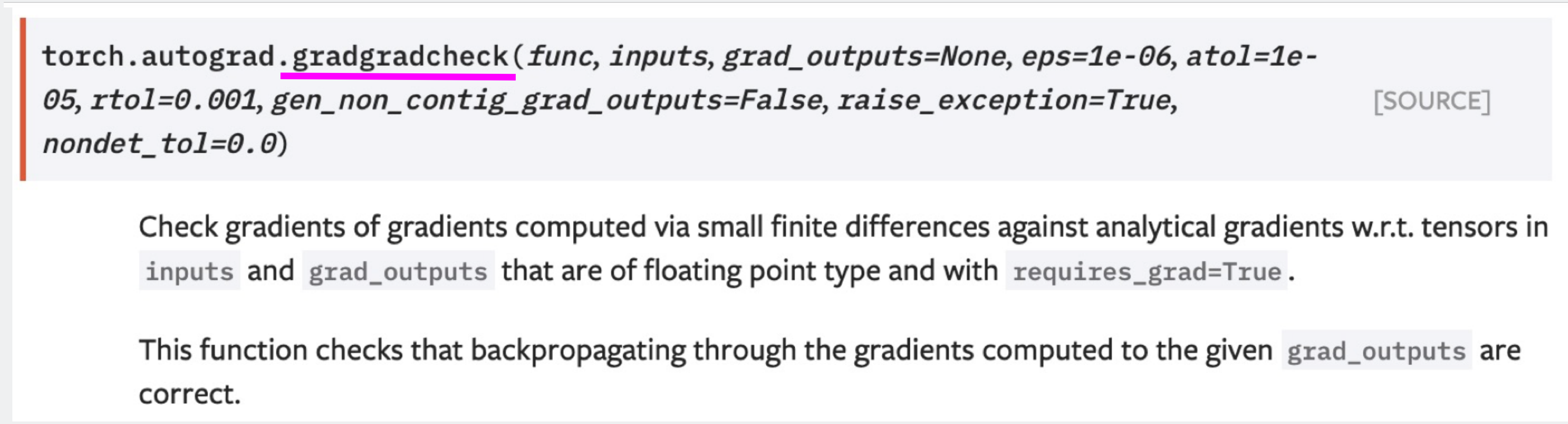
Check gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` that are of floating point type and with `requires_grad=True`.

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

Computing Gradients

① **Numeric gradient:** approximate, slow, easy to write

② **Analytic gradient:** exact, fast, error-prone



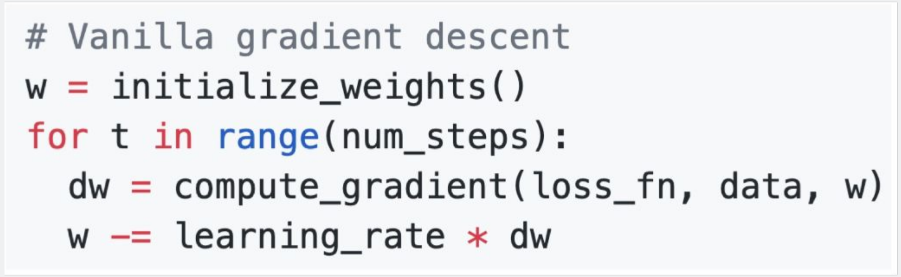
torch.autograd.gradgradcheck(*func*, *inputs*, *grad_outputs=None*, *eps=1e-06*, *atol=1e-05*, *rtol=0.001*, *gen_non_contig_grad_outputs=False*, *raise_exception=True*, *nondet_tol=0.0*) [SOURCE]

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

This function checks that backpropagating through the gradients computed to the given `grad_outputs` are correct.

Gradient Descent

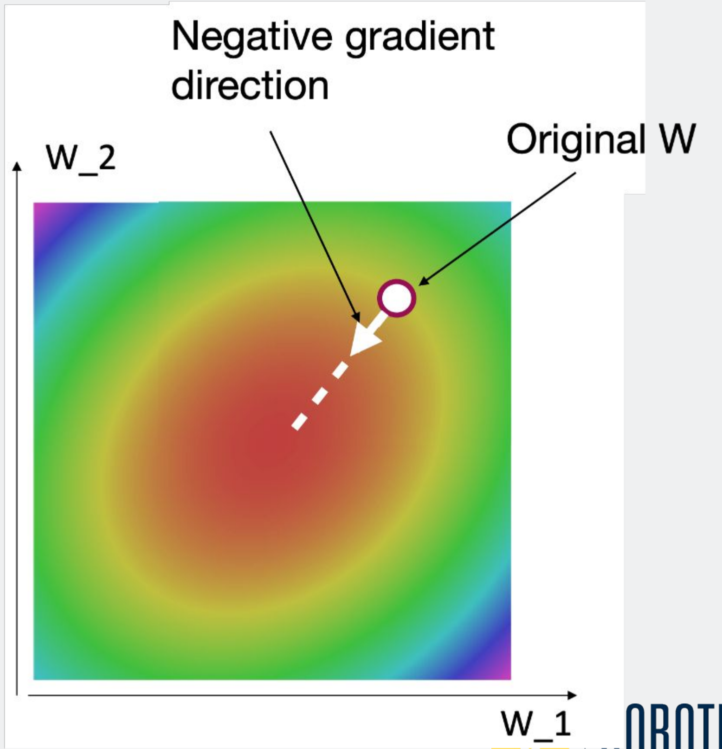
- Iteratively step in the direction of the negative gradient (direction of local steepest descent)



```
# 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



Q4: guarantee?

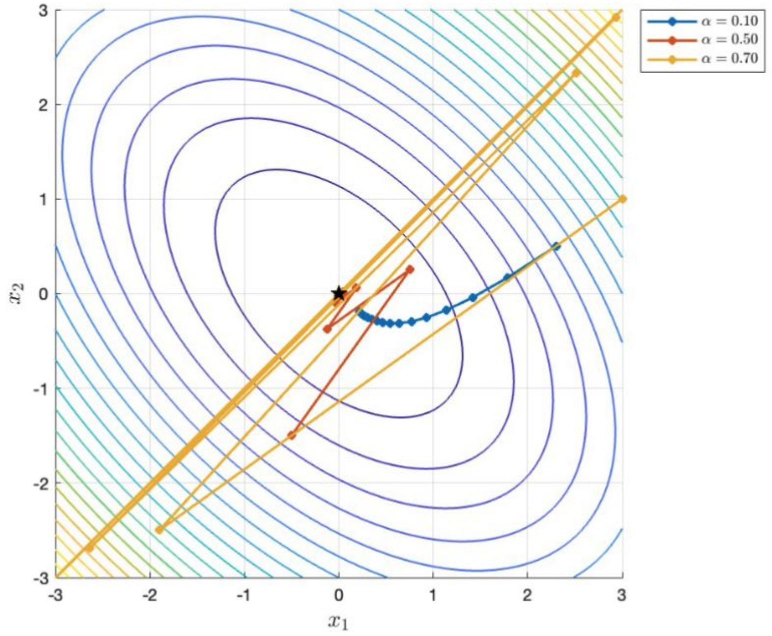
Aha Slides
(In-class participation)

https://ahaslides.com/J5D83

Q4

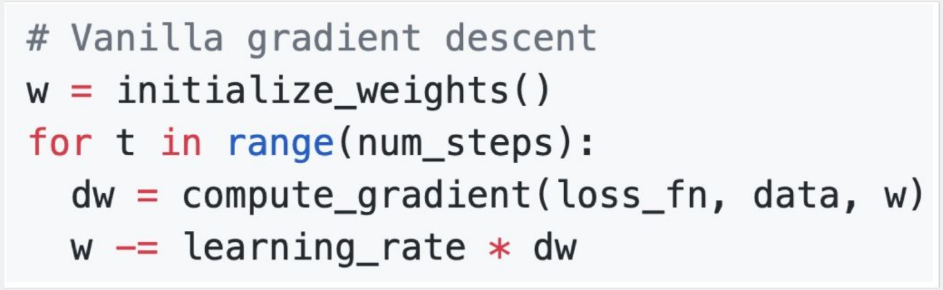
Gradient Descent on a 2D Function

- **Caution:** Gradient descent can be sensitive to the choice of step size α.
  - If α is too small, convergence will be slow.
  - If α is too large, the algorithm may overshoot or even diverge!



Gradient Descent

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



```
# 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

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

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

Full sum expensive when N is large!

Stochastic Gradient Descent (SGD)

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

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

```
# 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
```

Full sum expensive when N is large!

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

Hyperparameters:

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

Stochastic Gradient Descent (SGD)

$$L(W) = \mathbb{E}_{(x,y)\sim p_{data}}[L(x,y,W)] + \lambda R(W)$$

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

Think of loss as an expectation over the full **data distribution** p_{data}

Approximate expectation via sampling

$$\nabla_W L(W) = \nabla_W \mathbb{E}_{(x,y)\sim p_{data}}[L(x,y,W)] + \lambda R(W)$$

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

For reference: an interactive web demo:
http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

M | ROBOTICS

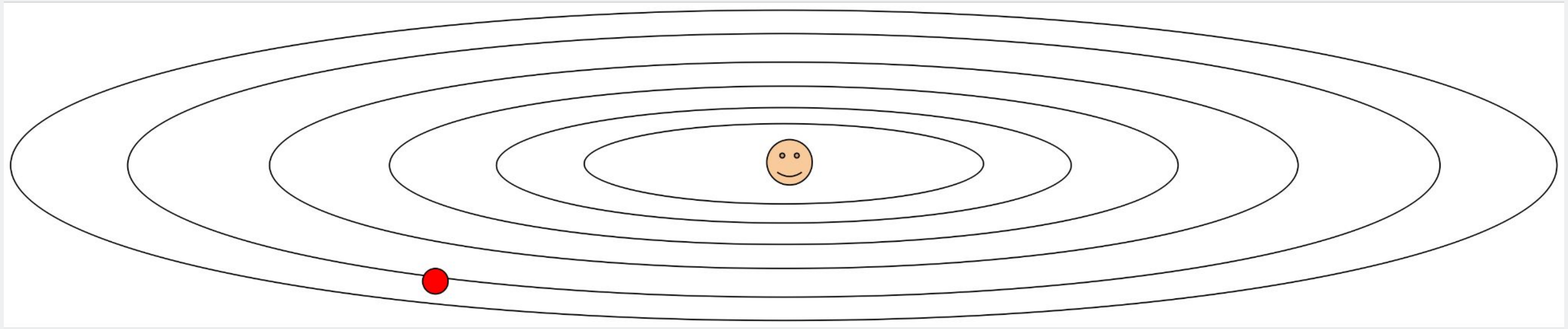
Aha Slides (In-class participation)

https://ahaslides.com/J5D83

Q5

Problem with SGD ①

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



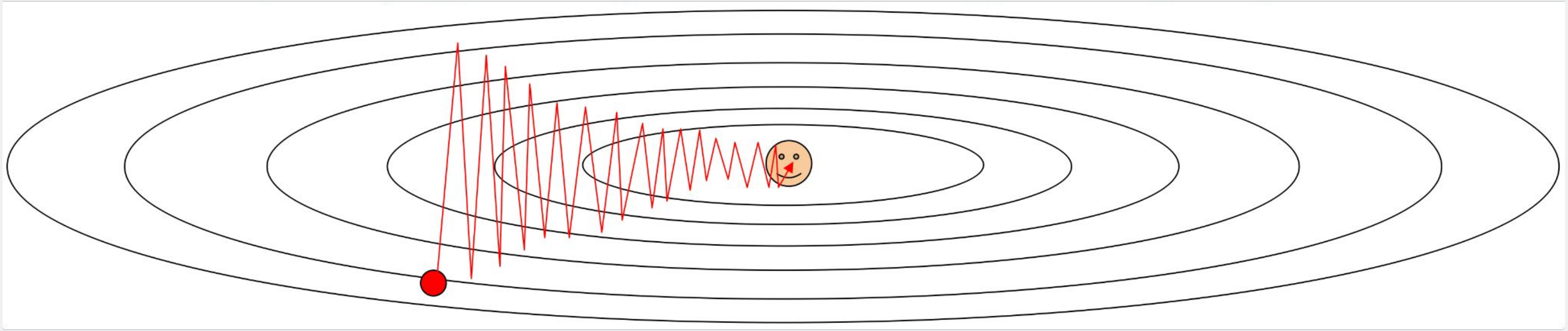
Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

Problem with SGD ①

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

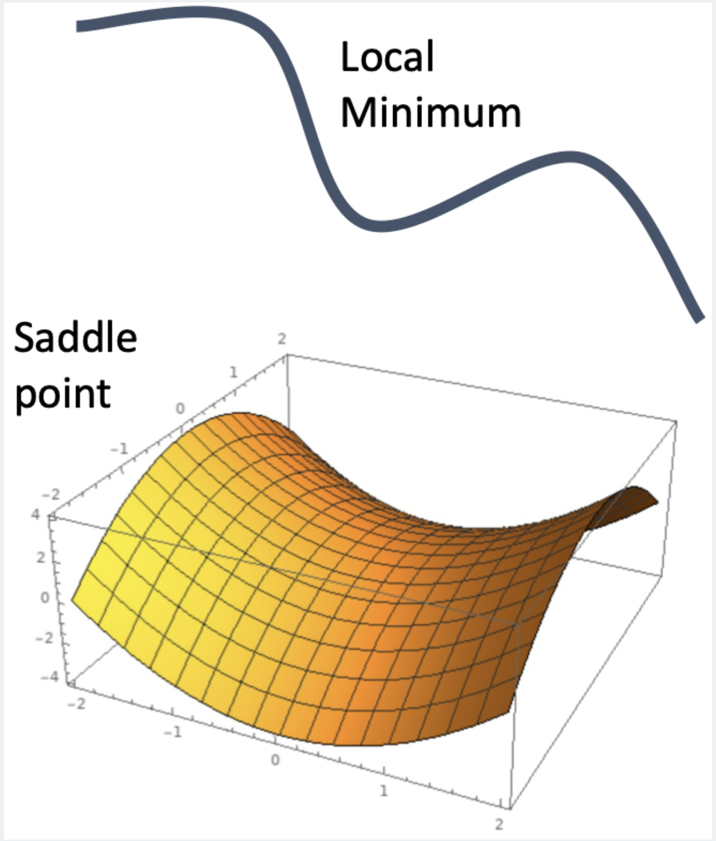
What does gradient decent do?

Very slow progress along shallow dimension, jitter along steep direction



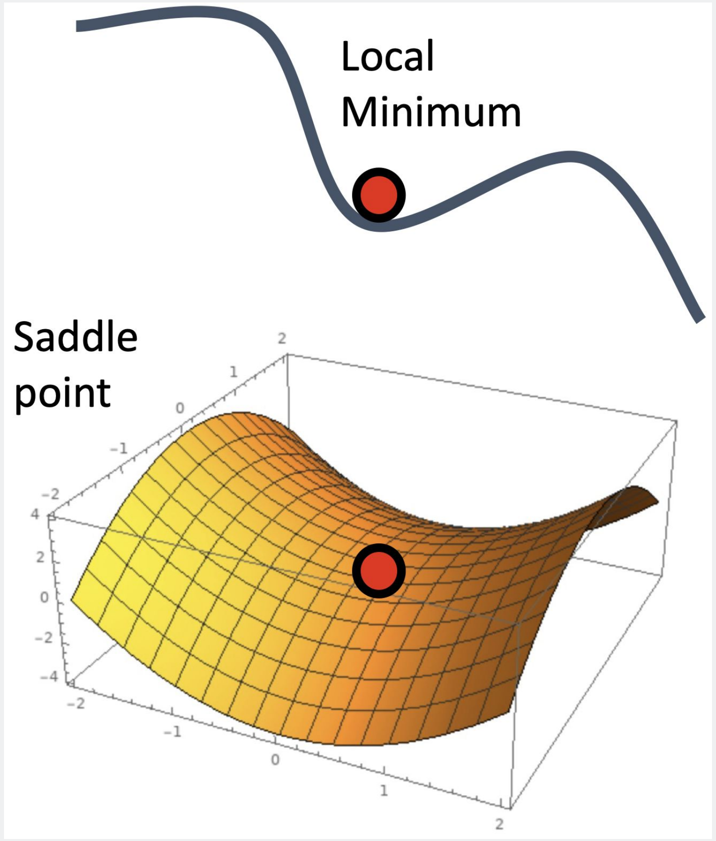
Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

Problem with SGD ②



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

Problem with SGD ②



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

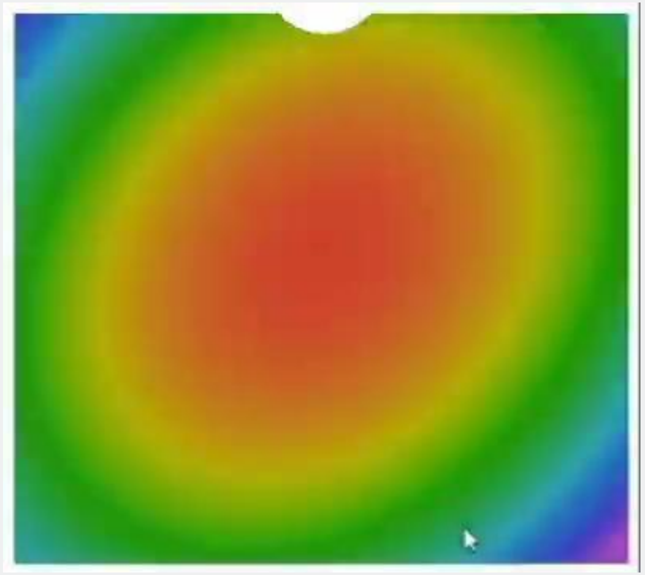
Zero gradient, gradient descent gets stuck

Problem with SGD ③

Our gradients come from mini batches so they can be noisy!

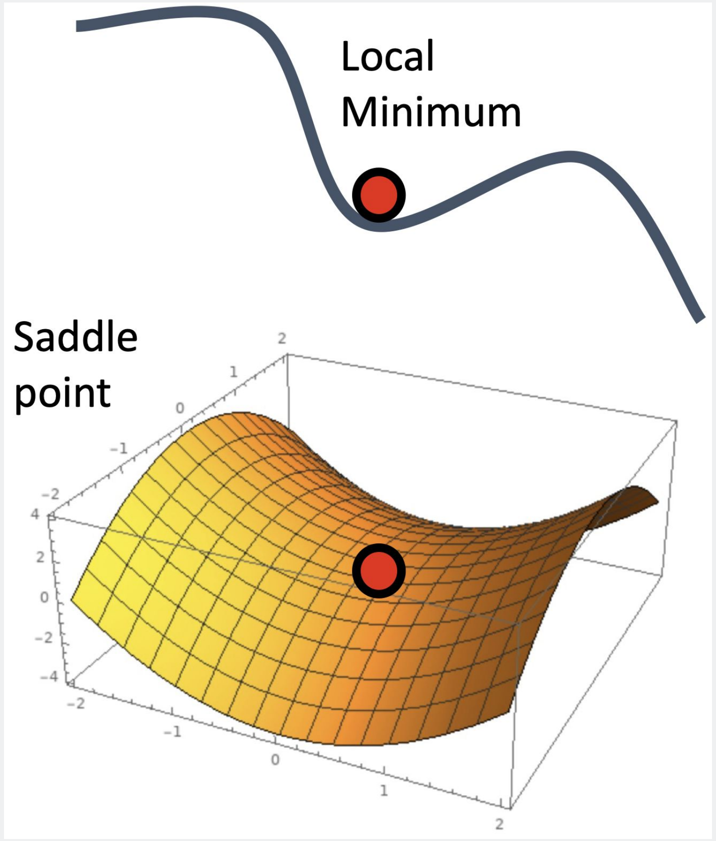
$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

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



M | ROBOTICS

Problem with SGD



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

Batched gradient descent always computes same gradients

SGD computes noisy gradients, may help to escape saddle points

More than SGD…

SGD + Momentum

"Ball running downhill"

SGD

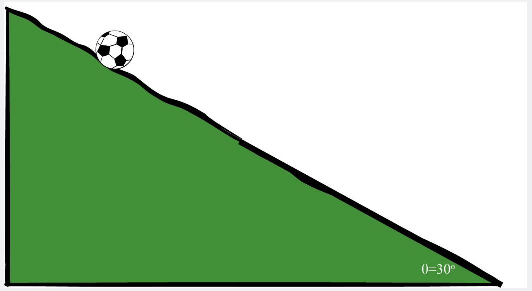
$$w_{t+1} = w_t - \alpha \nabla L(w_t)$$

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

SGD + Momentum

$$v_{t+1} = \rho v_t + \nabla L(w_t)$$
$$w_{t+1} = w_t - \alpha v_{t+1}$$

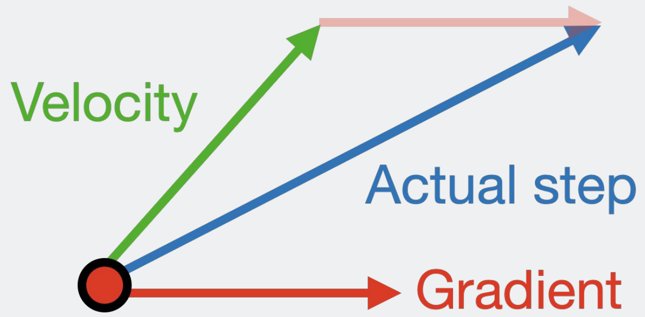
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```



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

SGD + Momentum

Momentum update:



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

SGD + Momentum

$$v_{t+1} = \rho v_t + \nabla L(w_t)$$
$$w_{t+1} = w_t - \alpha v_{t+1}$$

```
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

SGD + Momentum

SGD + Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla L(w_t)$$

$$w_{t+1} = w_t + v_{t+1}$$

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

SGD + Momentum

$$v_{t+1} = \rho v_t + \nabla L(w_t)$$

$$w_{t+1} = w_t - \alpha v_{t+1}$$

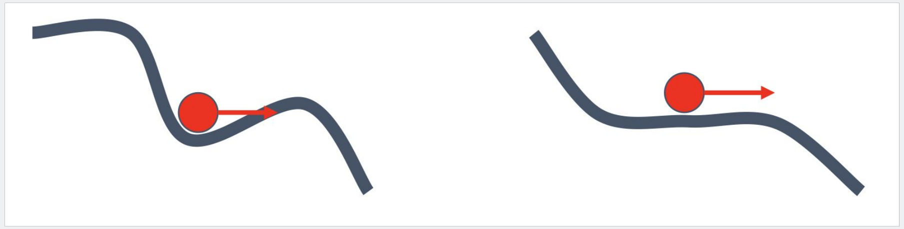
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

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

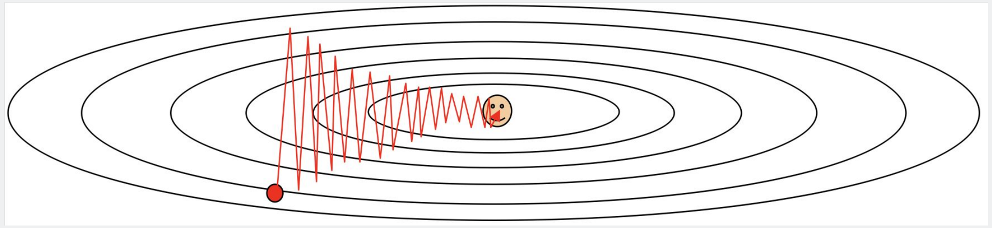
Sutskever et al, "On the importance of initialization and momentum in deep learning," ICML 2013

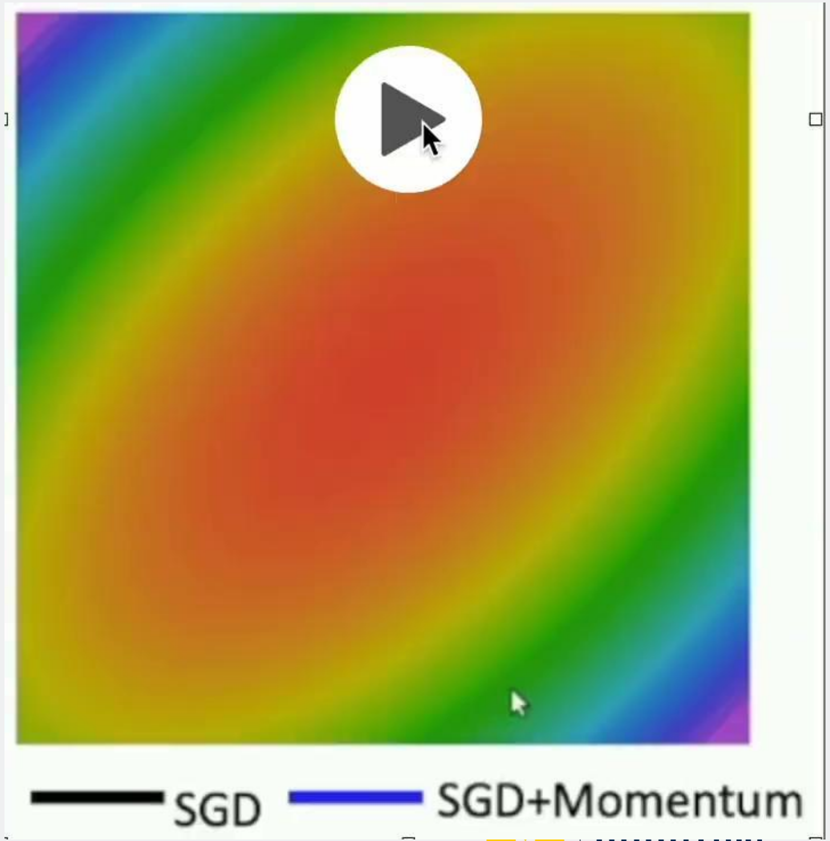
SGD + Momentum

Local Minima   Saddle Points



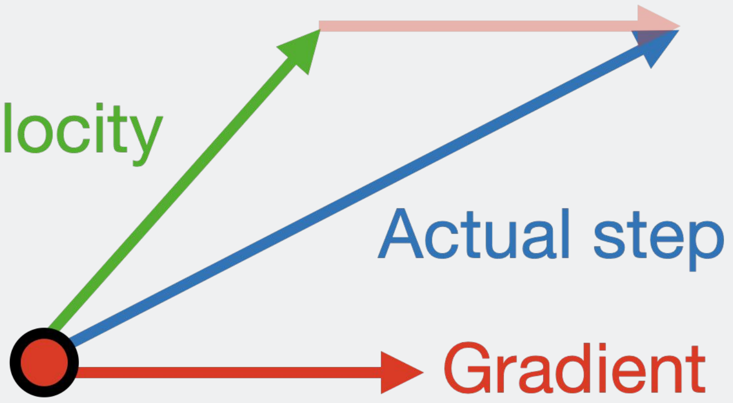
Poor Conditioning





SGD + Momentum

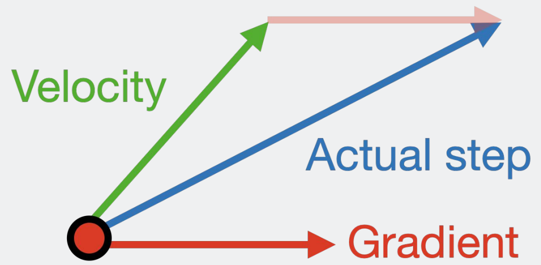
Momentum update:



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

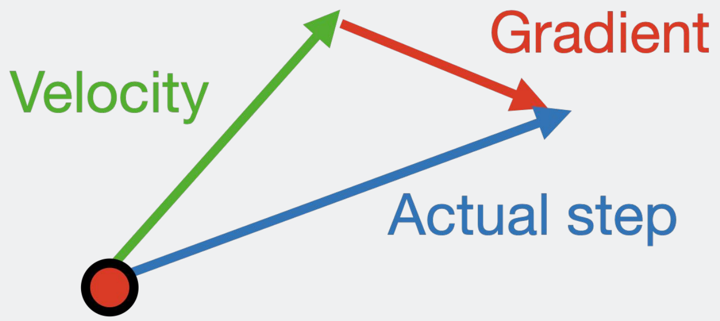
Nesterov Momentum

Momentum update:



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

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

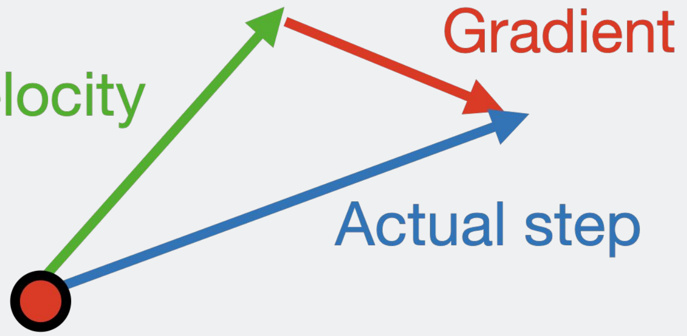
Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2),”, 1983”

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

Annoying, usually we want to update in terms of $w_t, \nabla L(w_t)$

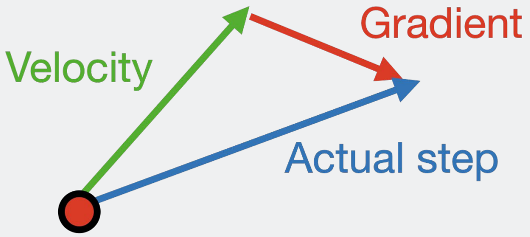


$$v_{t+1} = \rho v_t - \alpha \nabla L(w_t + \rho v_t)$$
$$w_{t+1} = w_t + v_{t+1}$$

"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

Annoying, usually we want to update in terms of $w_t, \nabla L(w_t)$



```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  old_v = v
  v = rho * v - learning_rate * dw
  w -= rho * old_v - (1 + rho) * v
```

$$v_{t+1} = \rho v_t - \alpha \nabla L(w_t + \rho v_t)$$

$$w_{t+1} = w_t + v_{t+1}$$

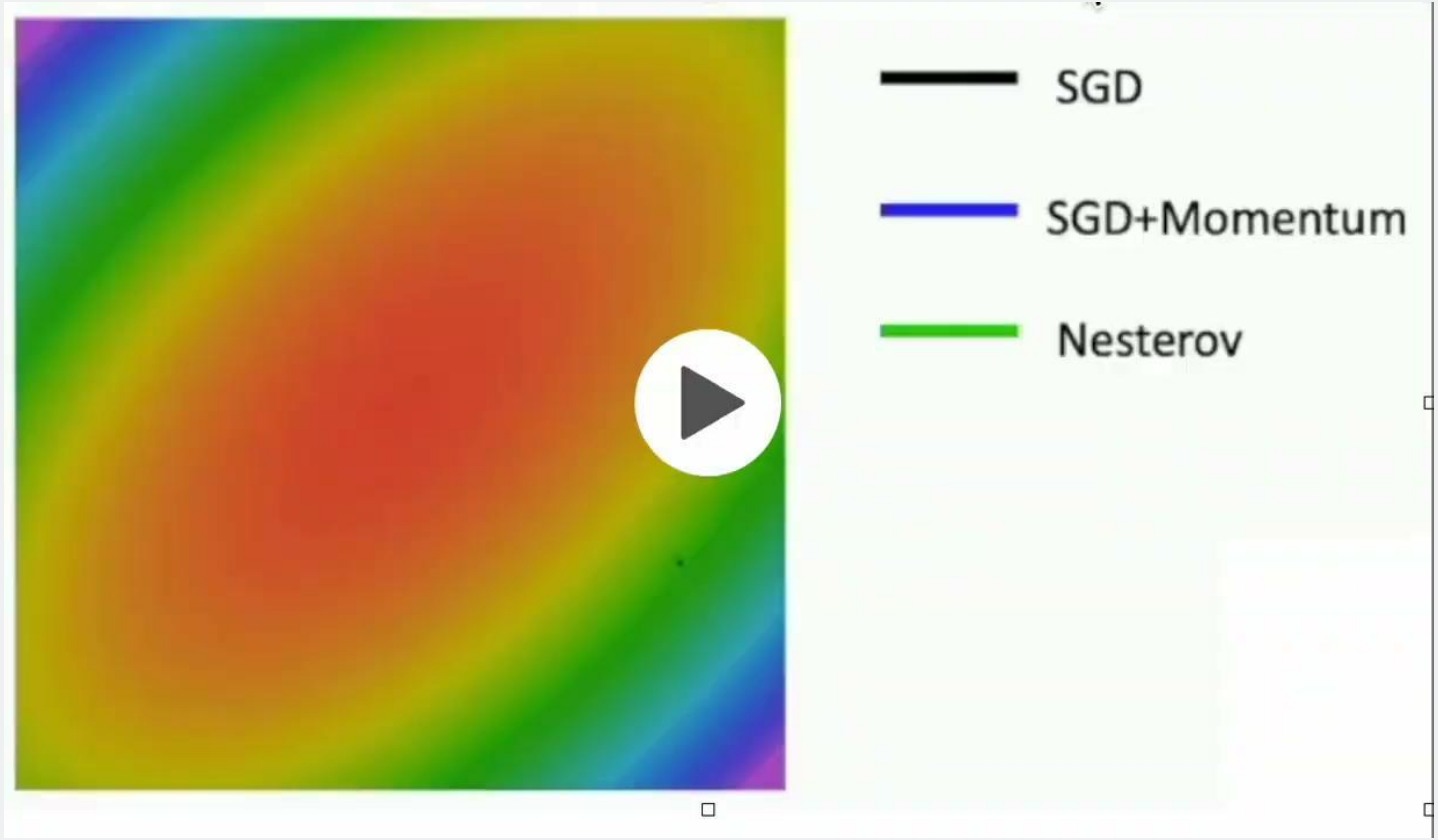
Change of variables and rearrange: $\tilde{w}_t = w_t + \rho v_t$

$$v_{t+1} = \rho v_t - \alpha \nabla L(\tilde{w}_t)$$

$$\tilde{w}_{t+1} = \tilde{w}_t - \rho v_t + (1+\rho) v_{t+1}$$

$$= \tilde{w}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

Nesterov Momentum



AdaGrad

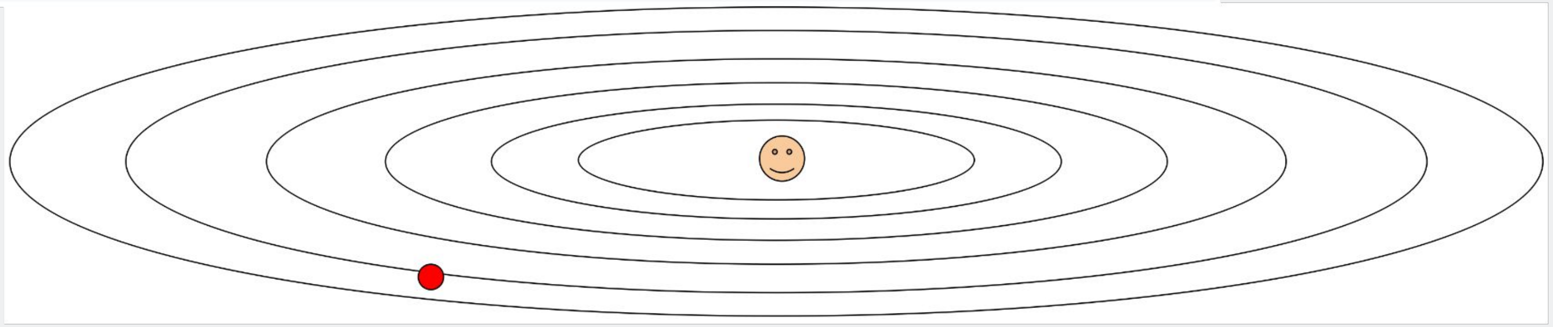
```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

- 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”

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Problem: AdaGrad will slow over many iterations



Q: What happens with AdaGrad?

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

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

RMSProp: “Leaky AdaGrad”

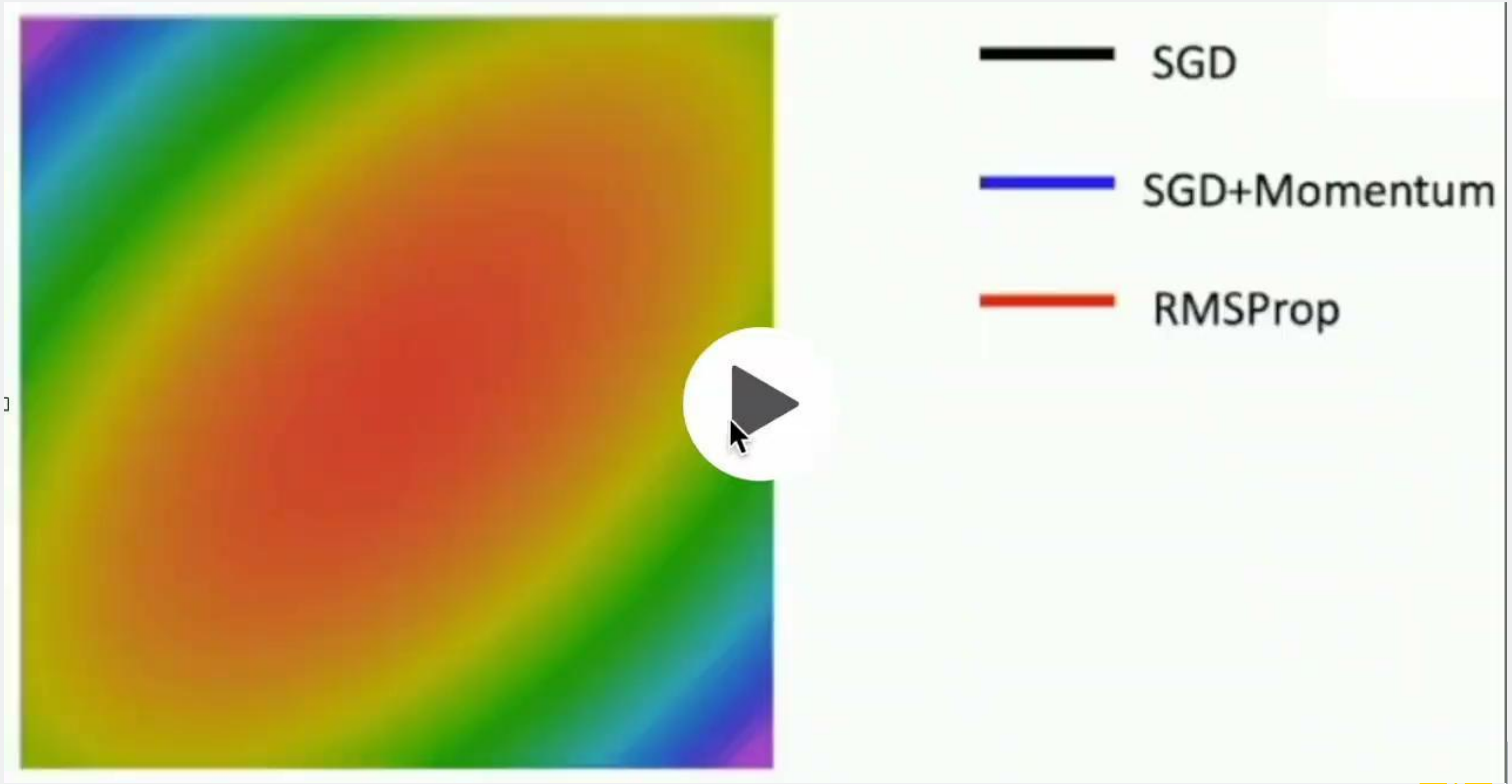
```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

RMSProp: “Leaky AdaGrad”



Due dates

Canvas Assignment: 20260121 Optimization Quiz

Scored - individual (as part of in-class activity points)

Due Sunday Jan. 25, 2026

P1 (KNN and Linear Classifier)

5 submissions per day - Start today!!!

Due Feb. 1, 2026