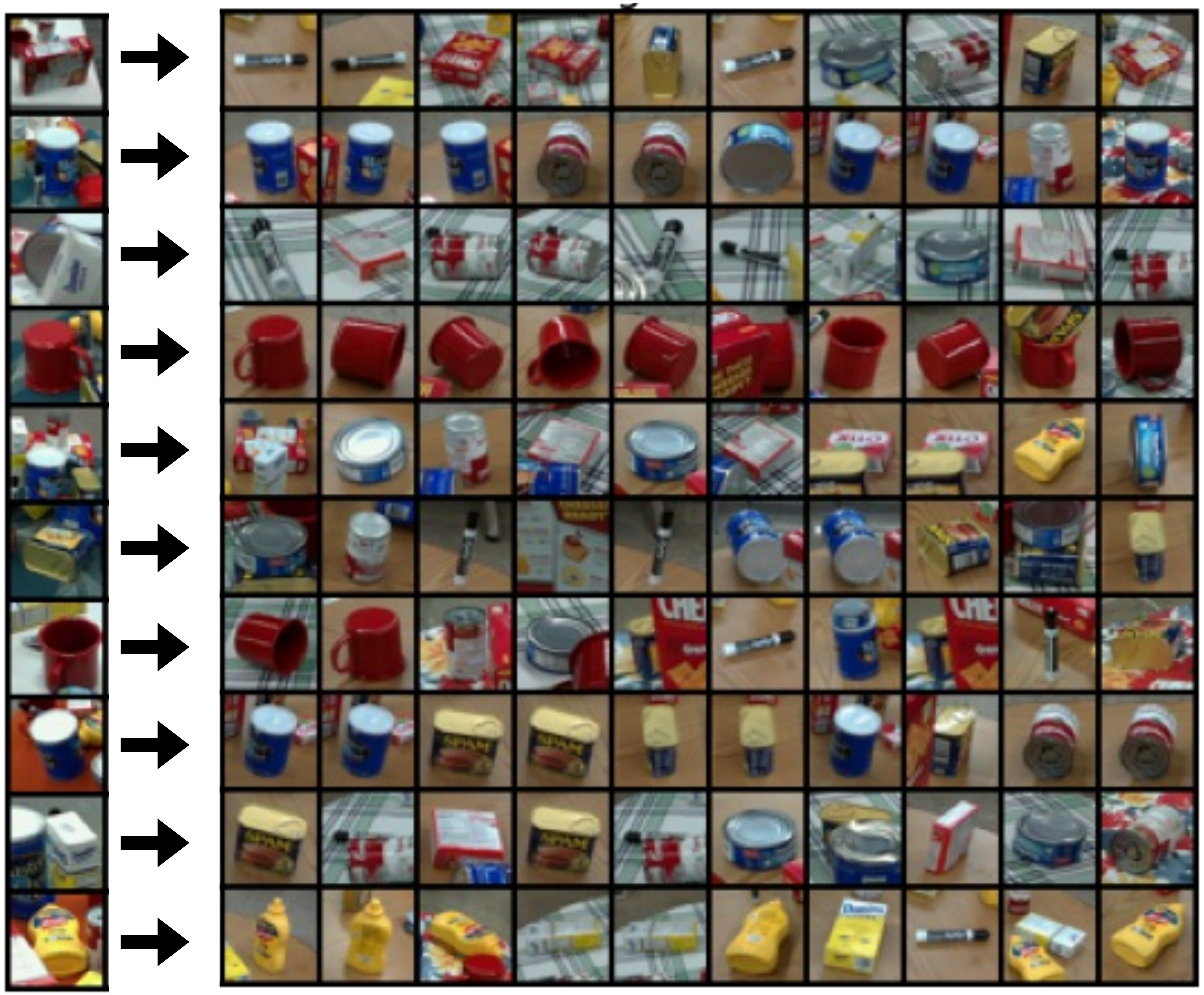
DeepRob

Lecture 3
Regularization + Optimization
University of Michigan | Department of Robotics

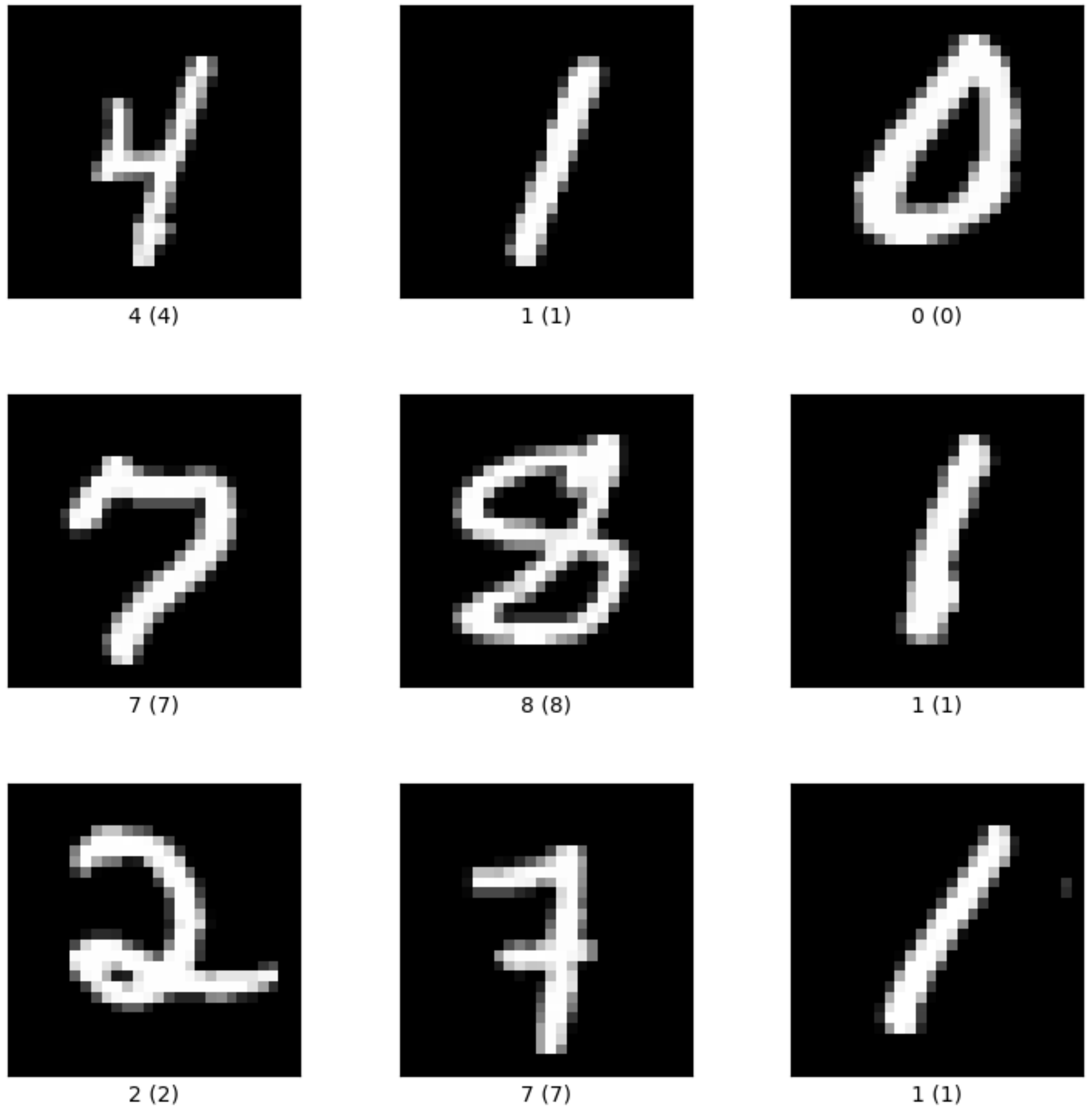
Recap: Image Classification

PROPS dataset



Recap: Image Classification

MNIST dataset

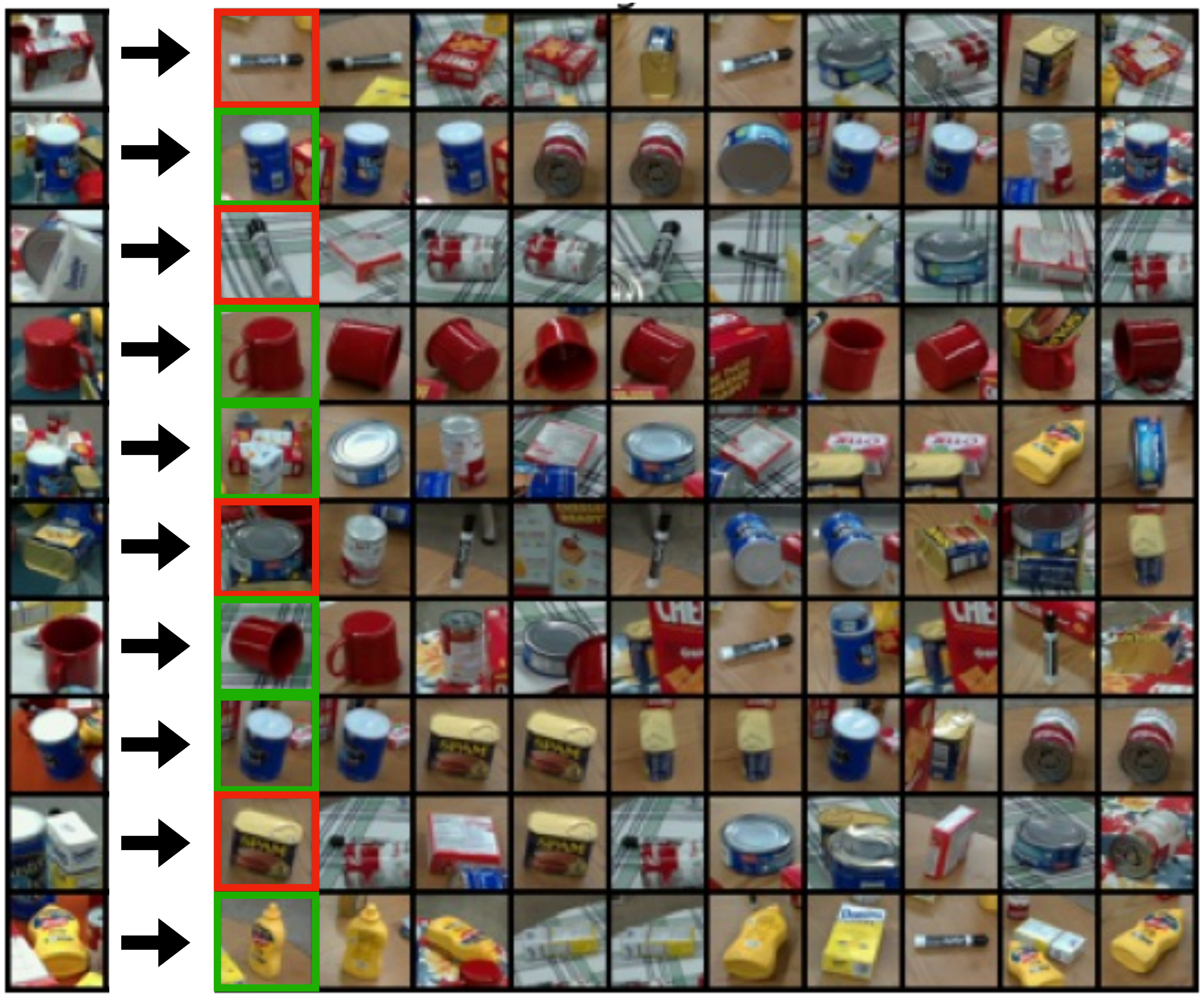


Labels

4	1	0
7	8	1
2	7	1

Recap: K Nearest Neighbor

PROPS dataset



KNN Pseudocode

1. Load training and testing data
2. Choose Hyperparameters (K=?)
3. For each point (image) in test data:

    find the distance to all training data points

    store the distance and sort it

    choose the first K points

    assign a class to the test image based on the majority of the classes

End

KNN – Some things to note

1. Hyperparameters: choose from k_choices

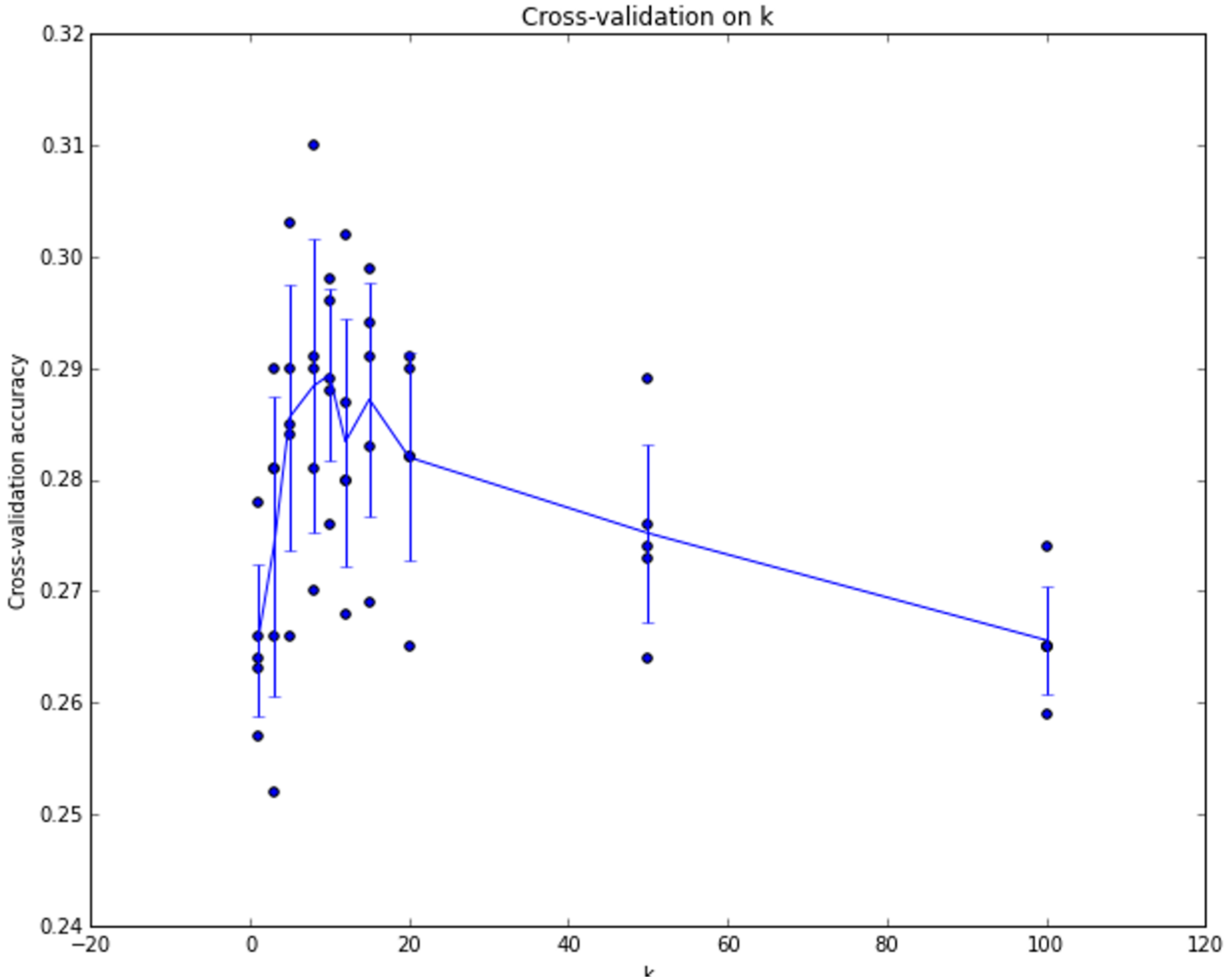
2. Cross-validation (e.g., 5-fold validation)

   First, split the data into **folds** `torch.chunk`

   Then, use all but one fold for train and one fold for validation

fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test

Setting Hyperparameters



Example of 5-fold cross-validation for the value of **k.**

Each point: single outcome.

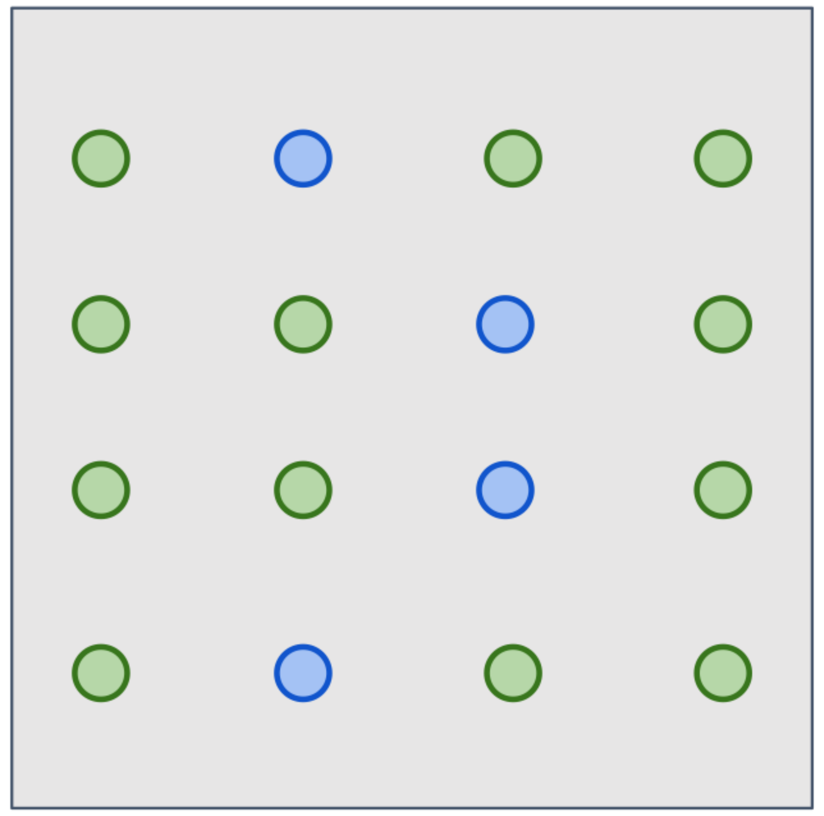
The line goes through the mean, bars indicated standard deviation

Problem—Curse of Dimensionality

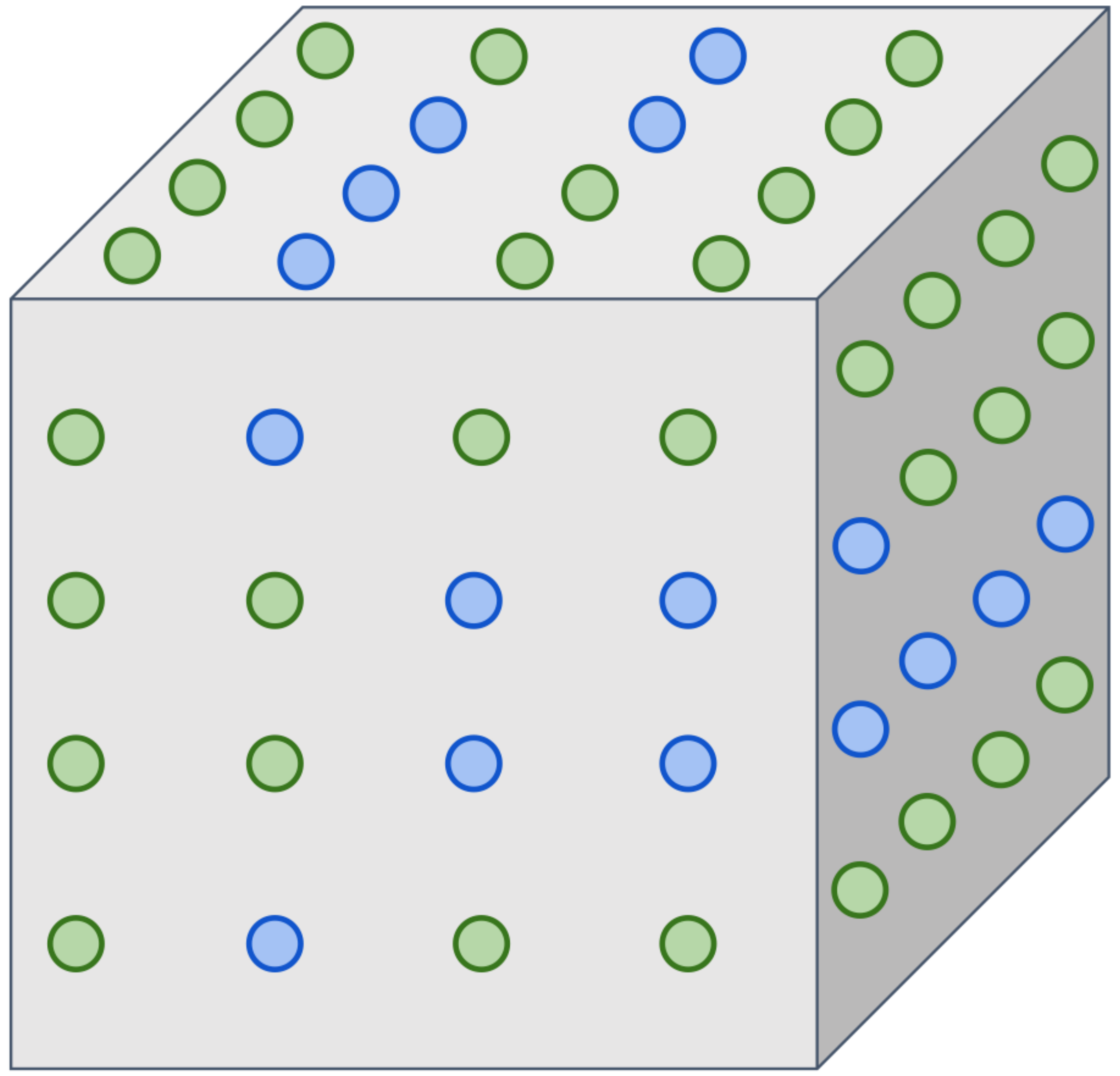
Curse of dimensionality: For uniform coverage of space, number of training points needed grows exponentially with dimension

Dimensions = 1
Points = 4

Dimensions = 2
Points = 4^2



Dimensions = 3
Points = 4^3



$$2^{32X32} \approx 10^{308}$$

K-Nearest Neighbors: Seldomly Used on Raw Pixels

Very slow at test time

Distance metrics on pixels are not informative

Original

Shifted

Tinted

Both images have same L2 distance to the original

Recap: Linear Classifier

Algebraic Viewpoint

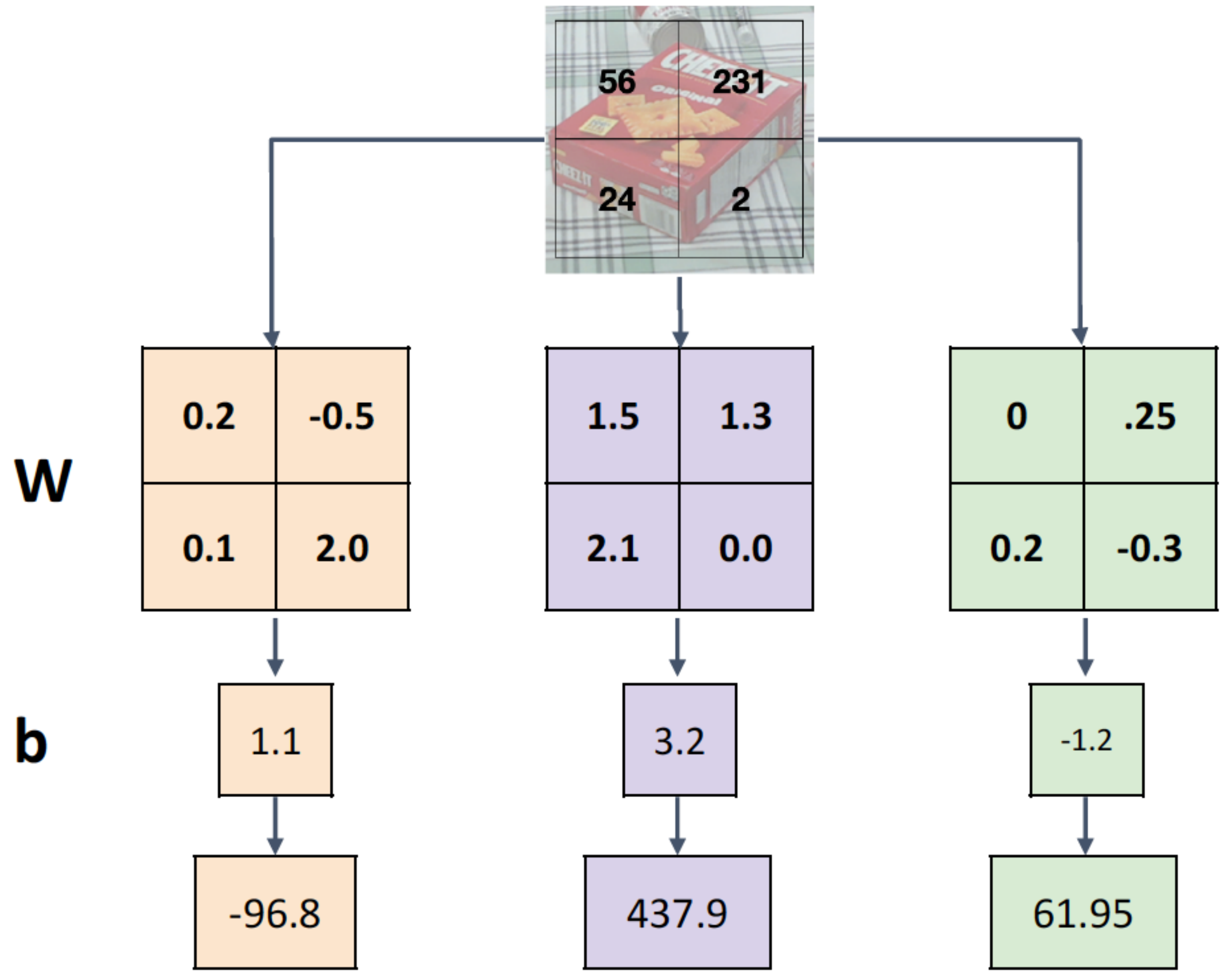
$$f(x,W) = Wx$$

Stretch pixels into column

Input image
(2, 2)

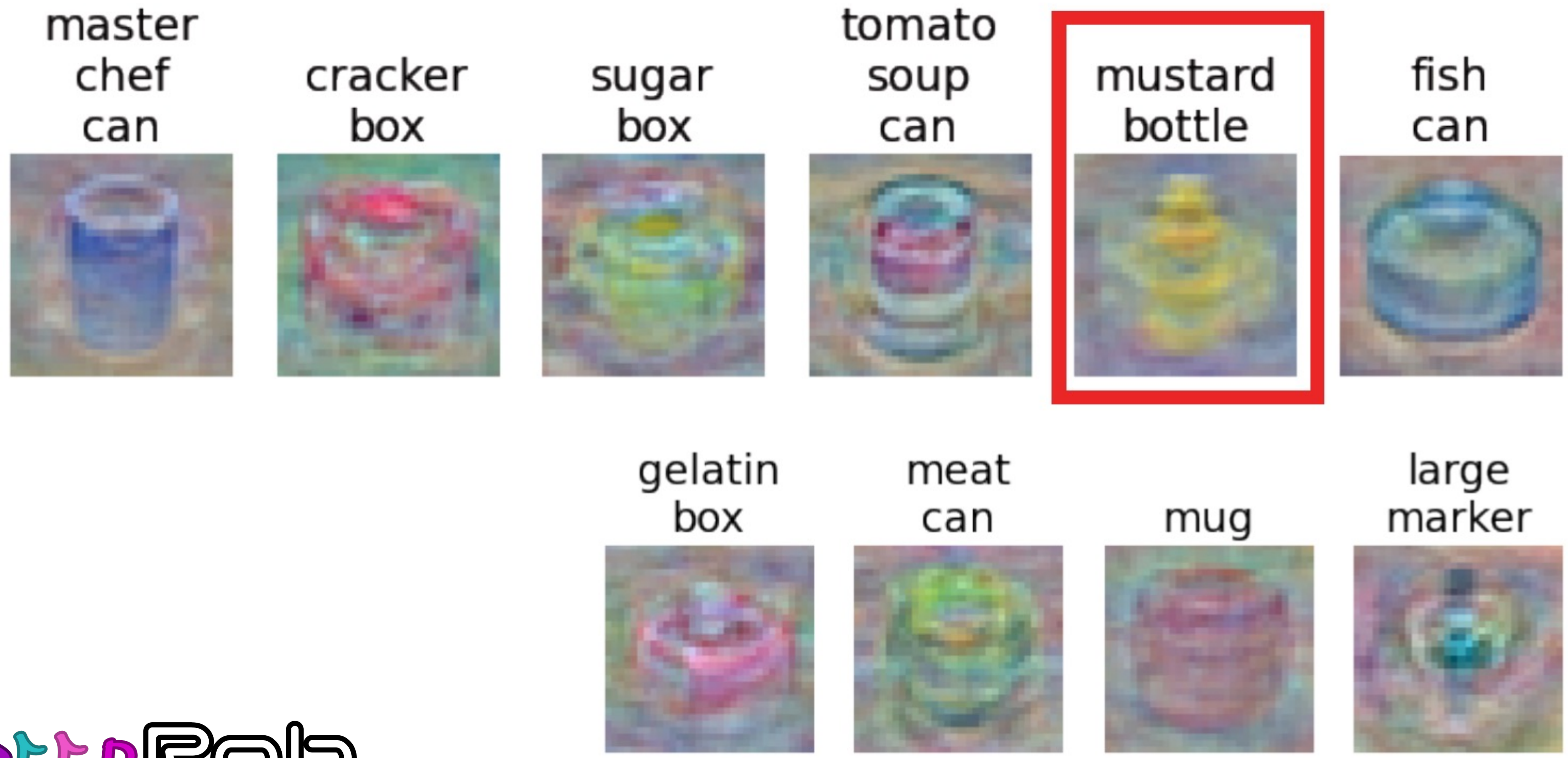
$$\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix}}_{W\ (3,\,4)} \underbrace{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}}_{(4,)} + \underbrace{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}}_{b\ (3,)} = \underbrace{\begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}}_{(3,)}$$

Recap: Linear Classifier



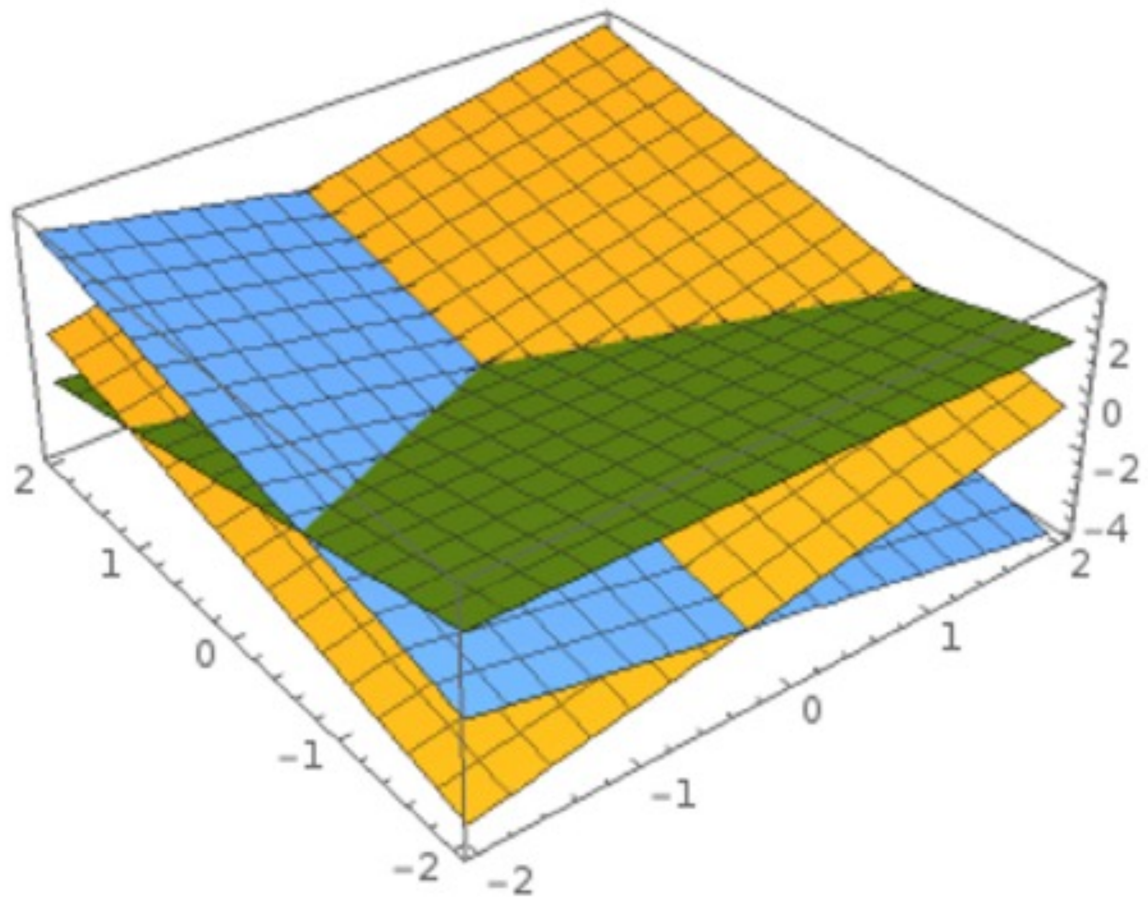
Visual Viewpoint

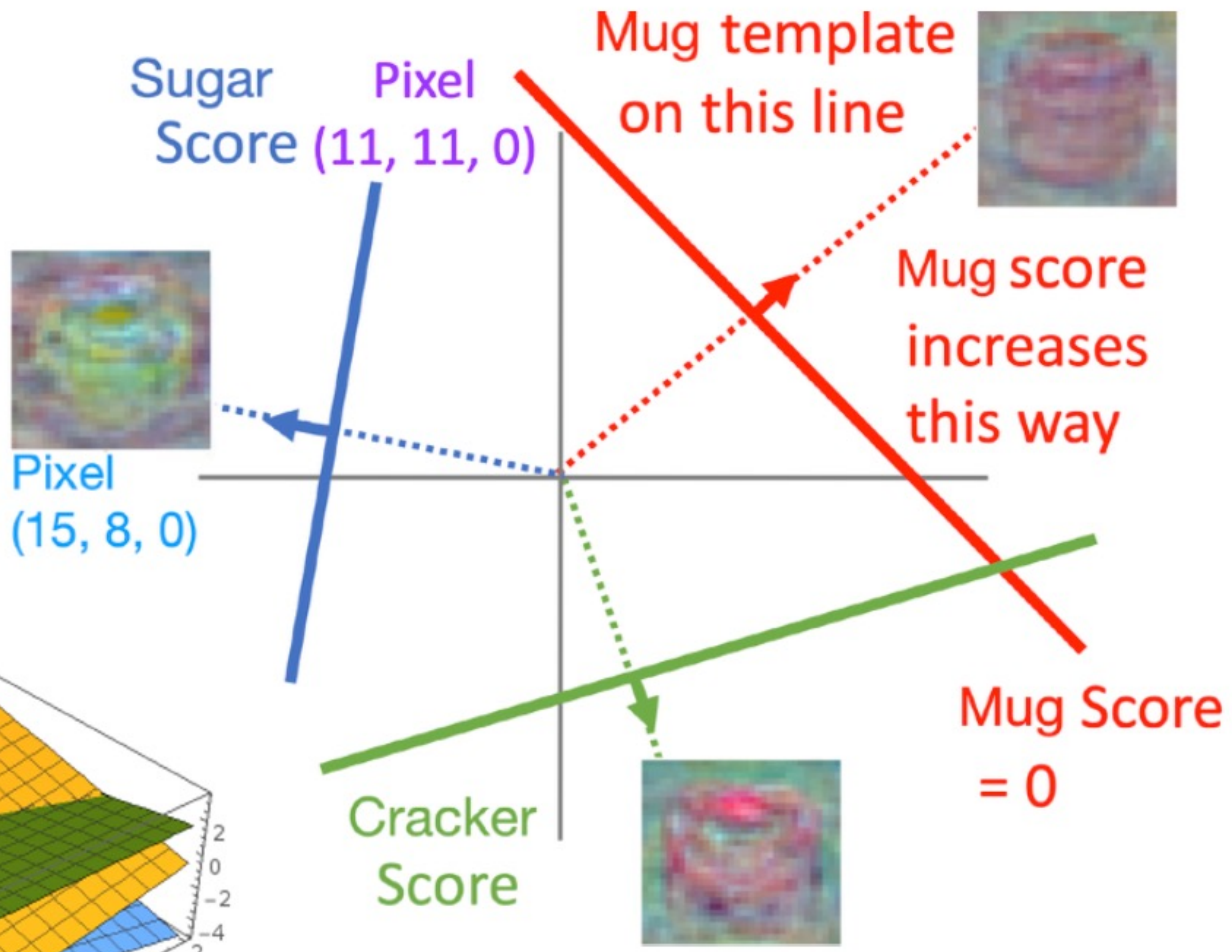
"stretch rows of W into images"



Recap: Linear Classifier

Geometric Viewpoint



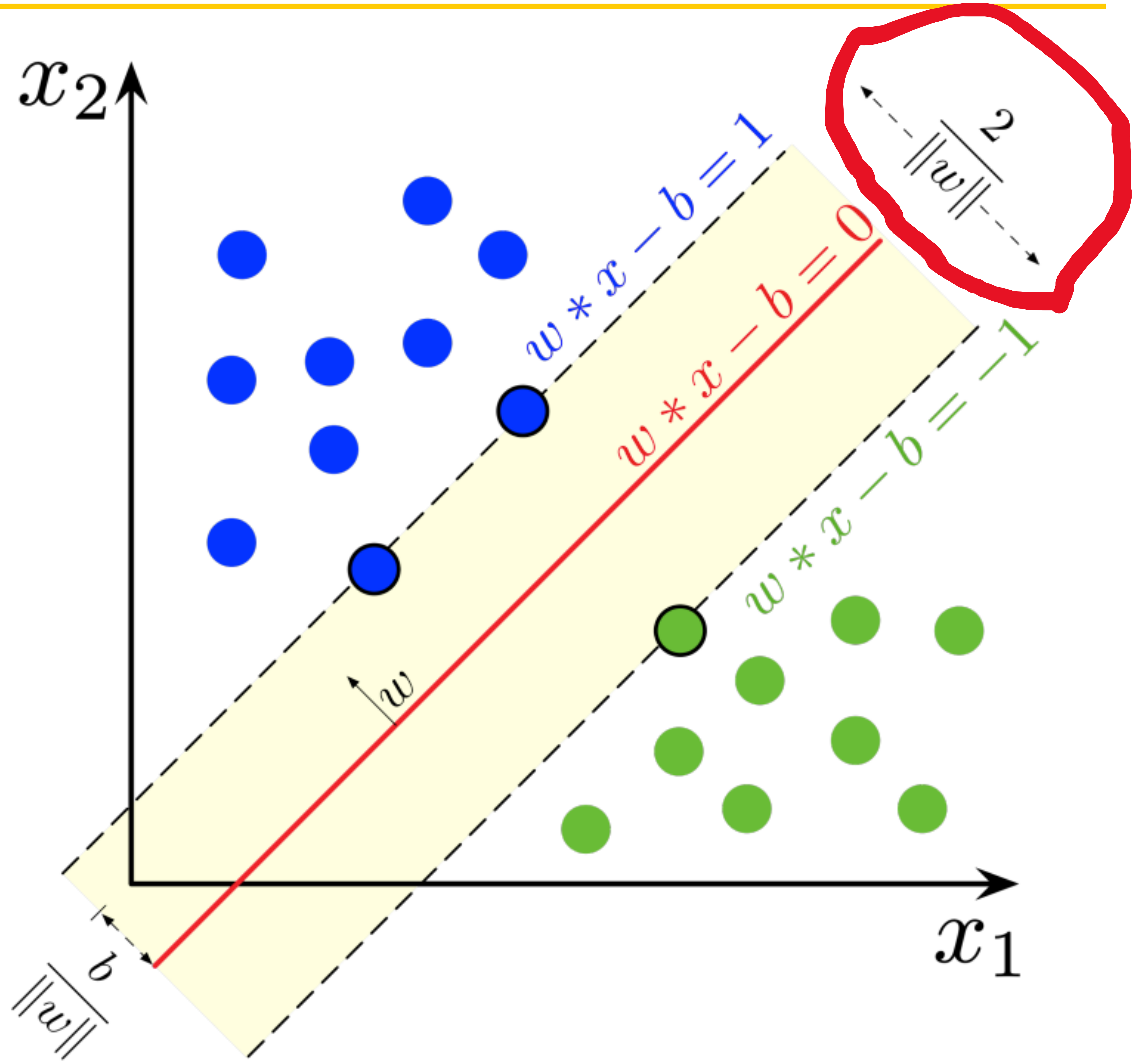


Recap—Linear SVM

Training Data

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$$

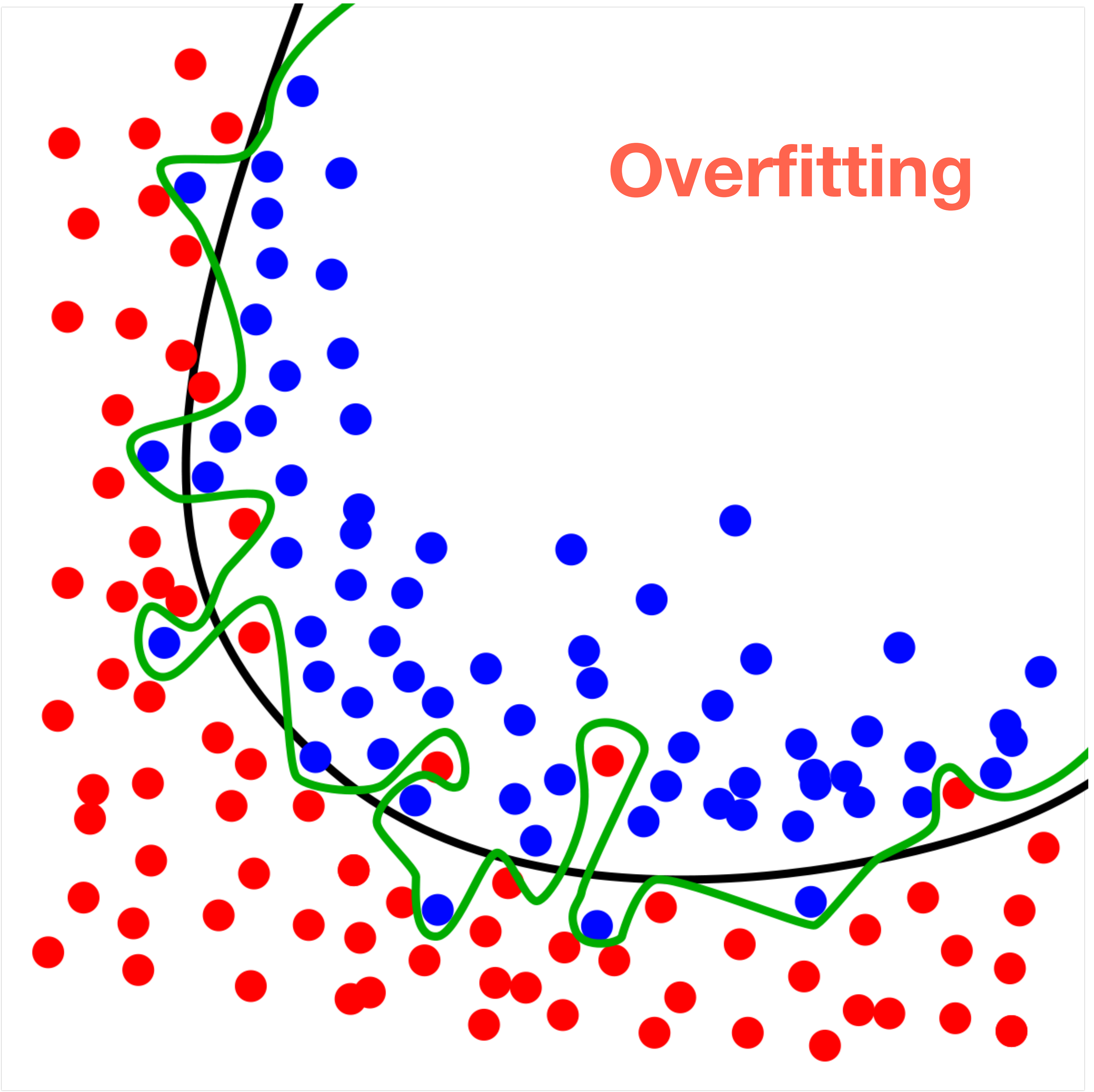
Hyperplane $\boxed{\mathbf{w}^{\mathrm{T}}\mathbf{x} - b = 0}$



What if there are misclassifications?

Hinge Loss (soft margin)

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



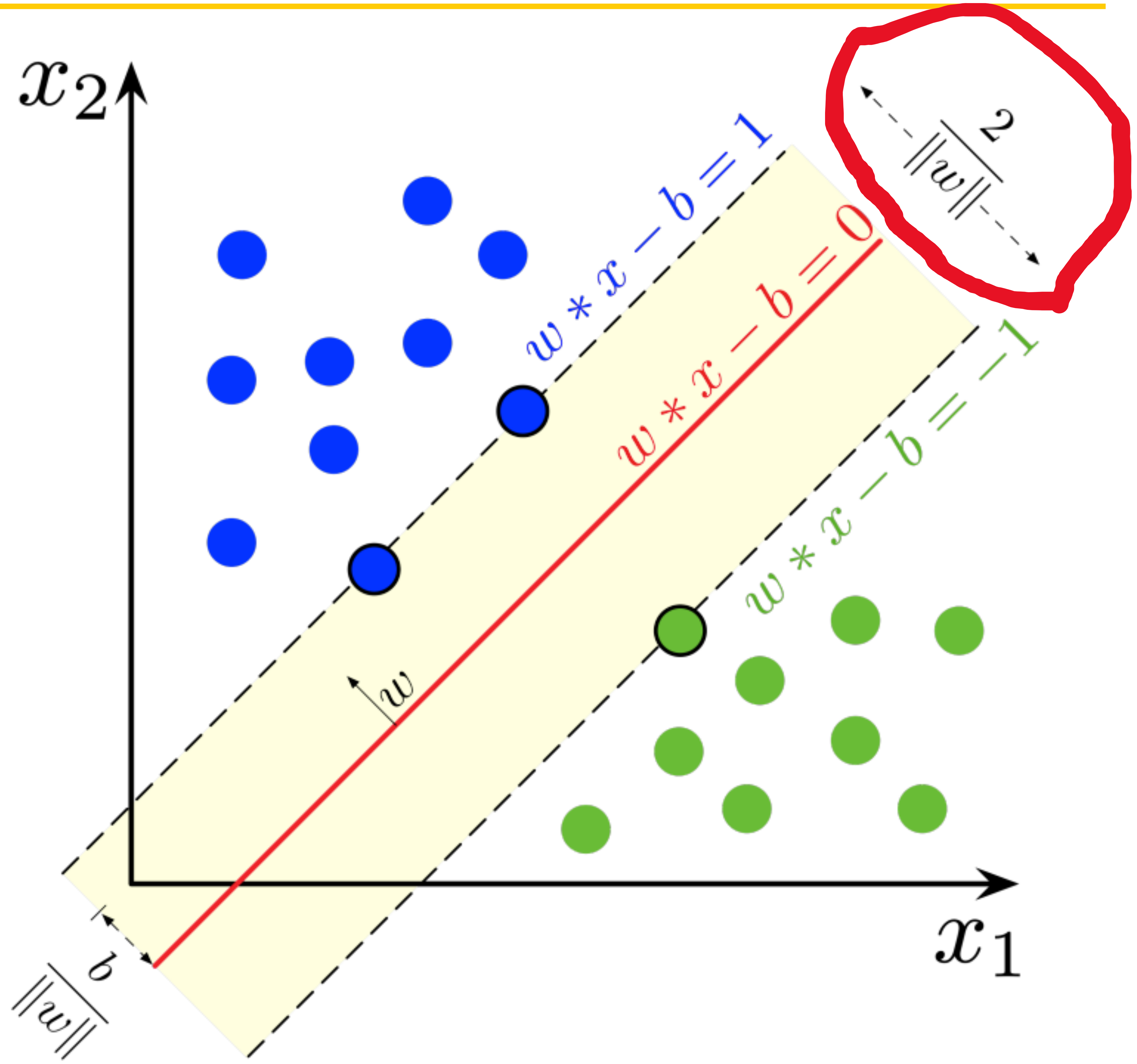
Back to SVM...

Training Data

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$$

Hyperplane $\boxed{\mathbf{w}^{\mathrm{T}}\mathbf{x} - b = 0}$

Maximize $\frac{2}{\|w\|}$ ➡ Minimize $\frac{\|w\|}{2}$



Loss Functions Quantify Preferences

Q: How do we find the best W,b?

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

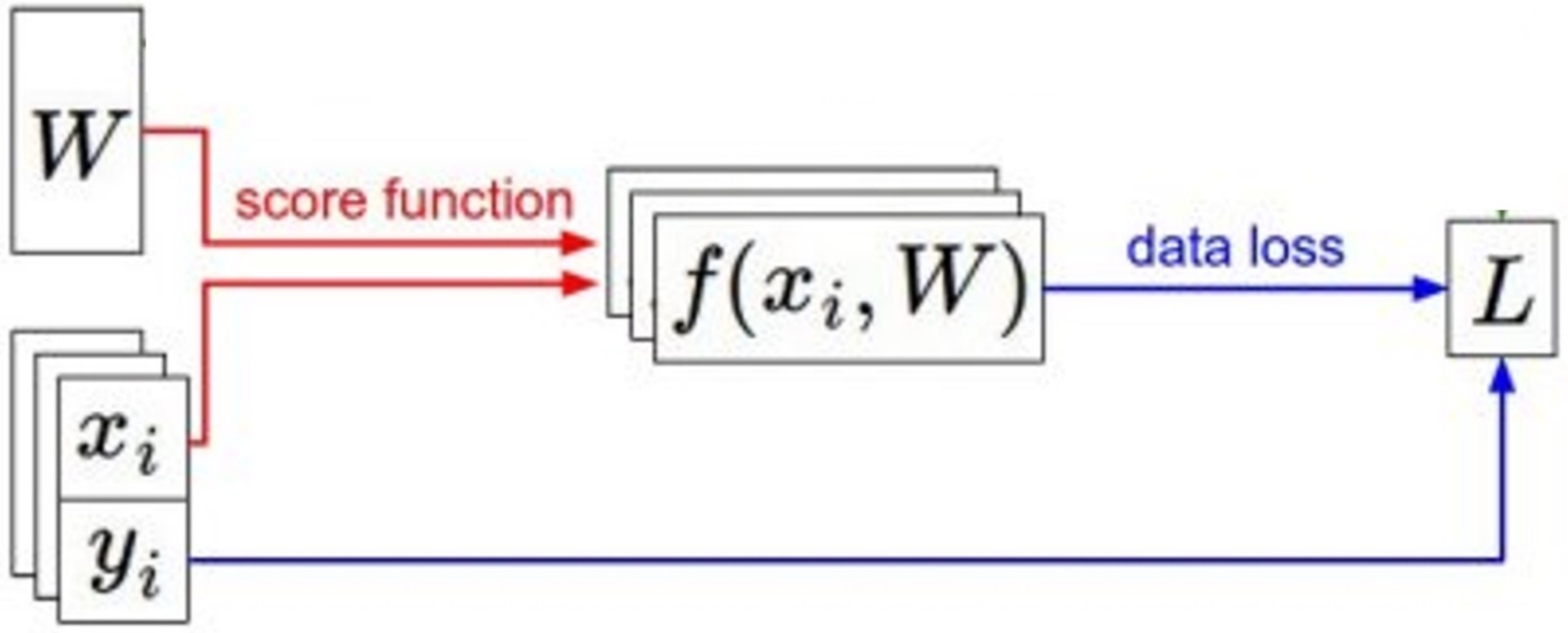
$$s = f(x; W, b) = Wx + b$$

Linear classifier

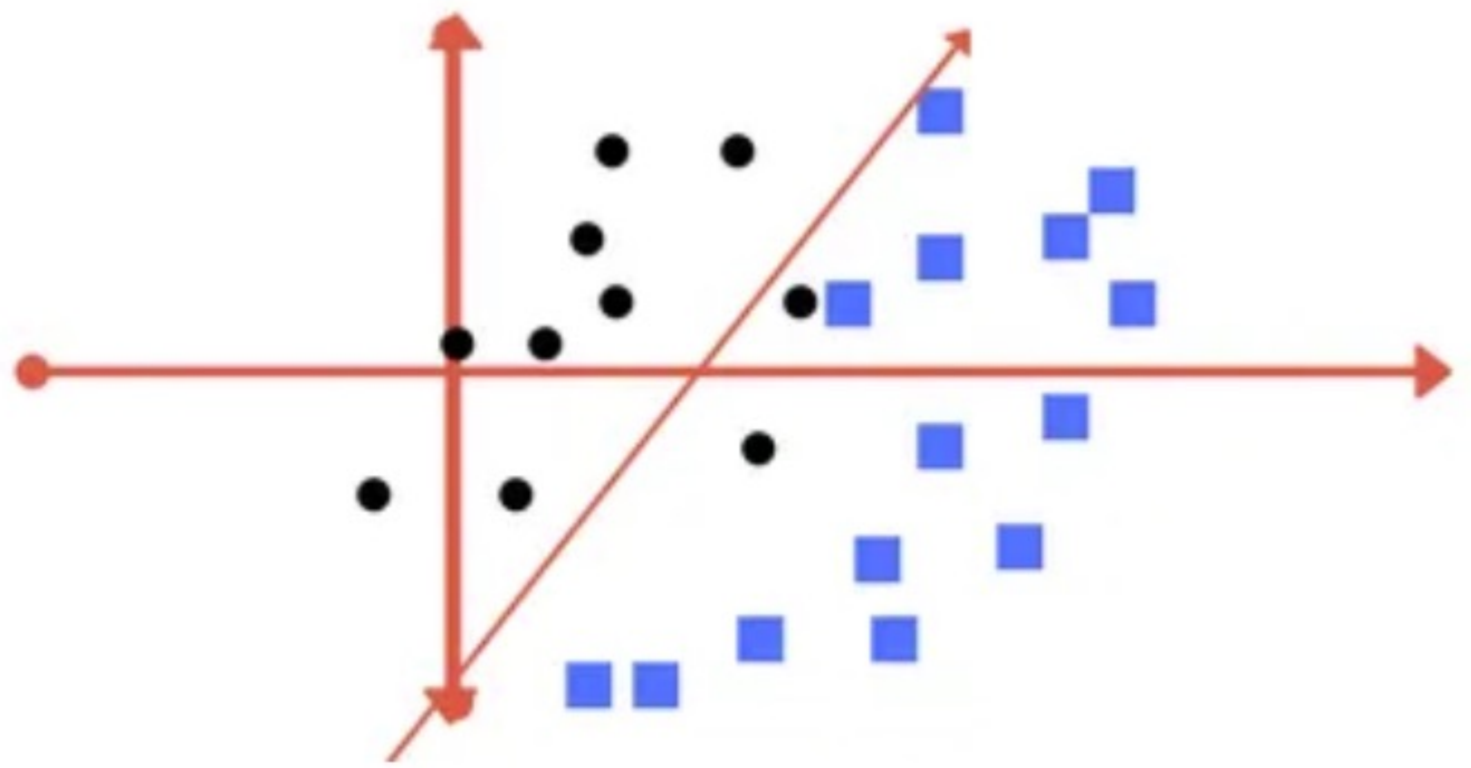
Softmax: $L_i = -\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right)$

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

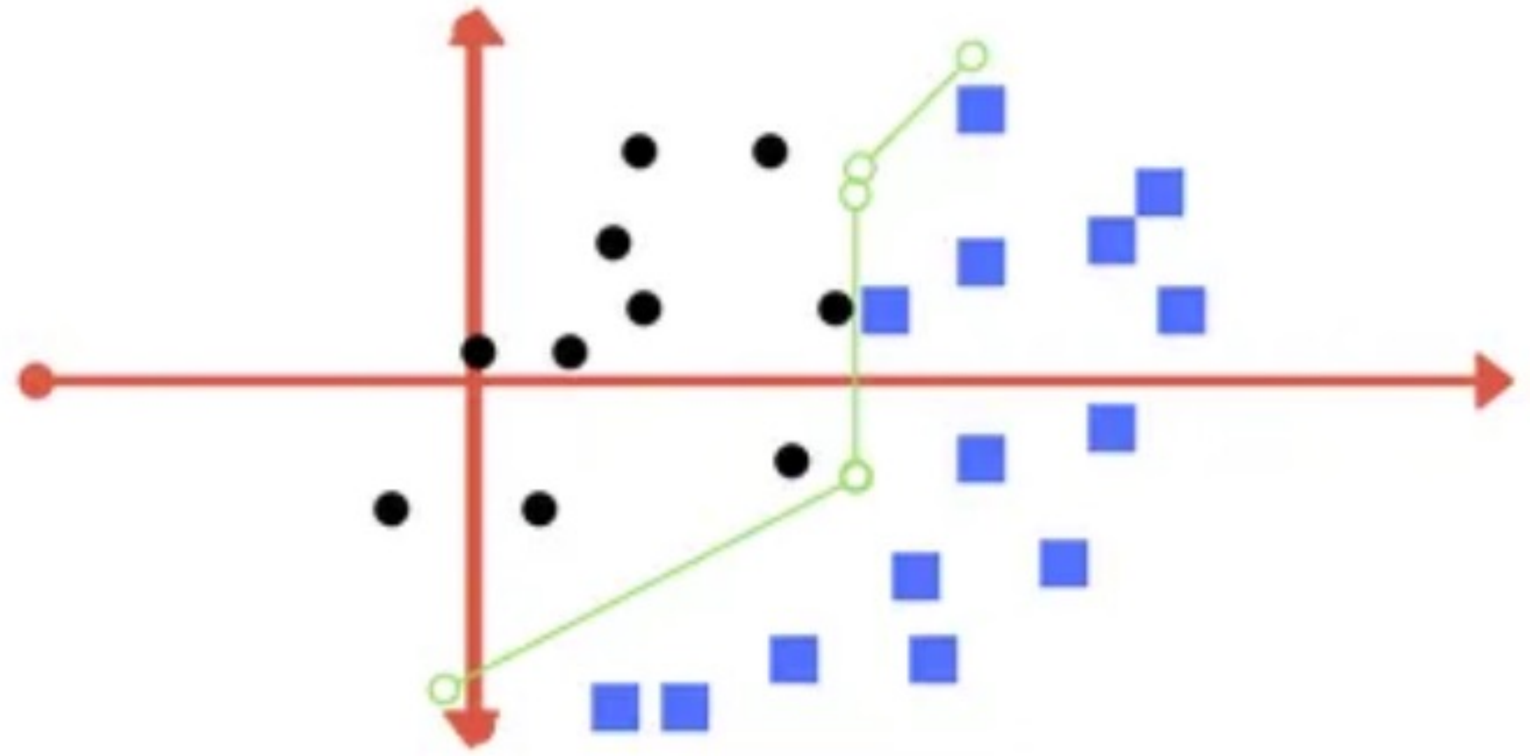
$+\lambda \frac{\|w\|}{2}$



Loss Functions Quantify Preferences



Q: Low or High regularization?



Q: Low or High regularization?

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

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

$+C\frac{\|w\|\text{^}2}{2}$

General Case: Adding Regularization Term

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

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

Regularization: Example

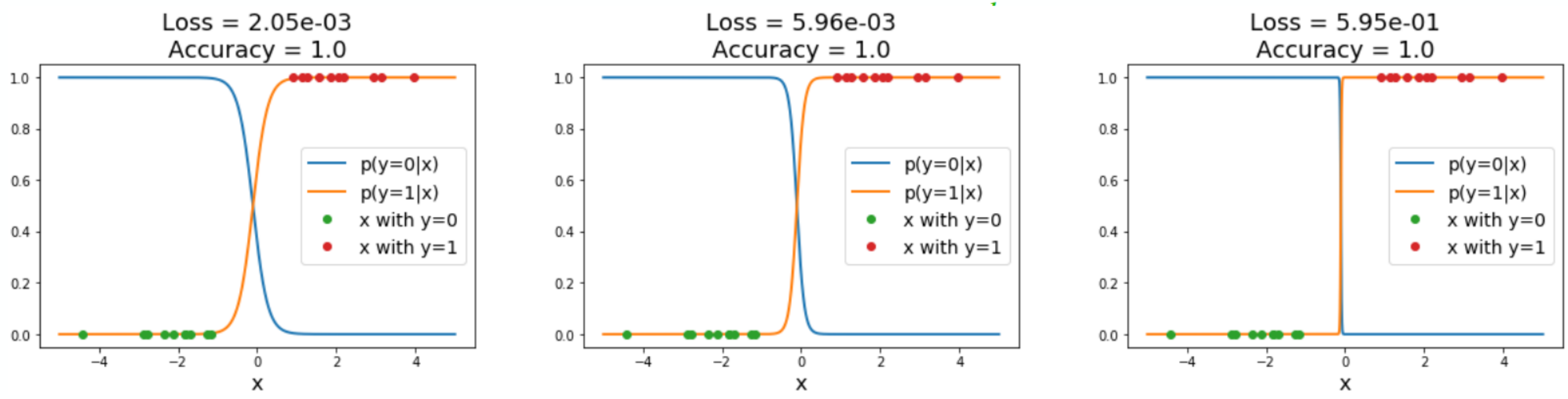
Example: Linear classifier with 1D inputs, 2 classes, and softmax loss

$$s_i = w_i x + b_i$$

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

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

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



Regularization: Expressing Preference

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

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

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

L2 Regularization

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

L2 Regularization prefers weights to be "spread out"

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

Same predictions, so data loss will always be the same

How to find 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

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

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!
not bad but not great…   (SOTA is ~95%)

Idea #2: Follow the slope

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

Idea #2: Follow the slope

"gradient descent"

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

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

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**.

Example:

Current W:

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

loss 1.25347

Gradient $\frac{dL}{dW}$

[?,
?,
?,
?,
?,
?,
?,
?,
?, …]

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 $\frac{dL}{dW}$

[-2.5,
?,
?,

(1.25322 - 1.25347)/
0.0001
= -2.5

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

?, …]

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 $\frac{dL}{dW}$

[-2.5,
0.6,
?,
?,

(1.25353 - 1.25347)/
0.0001
= 0.6

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

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**.

```
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.
  """
```

Computing Gradients

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

- **Numeric gradient:** approximate, slow, easy to write
- **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.

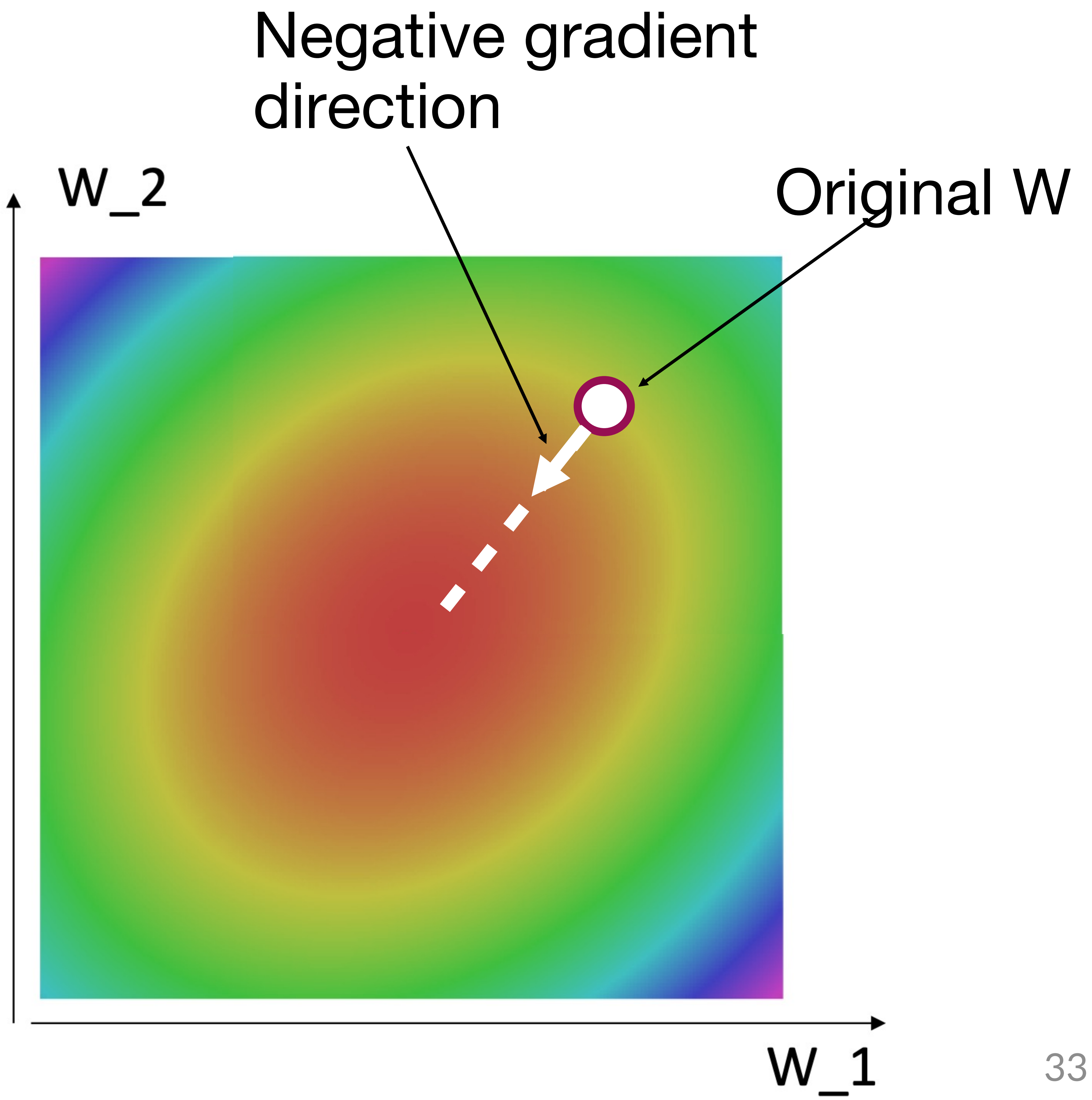
Gradient Descent

- 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



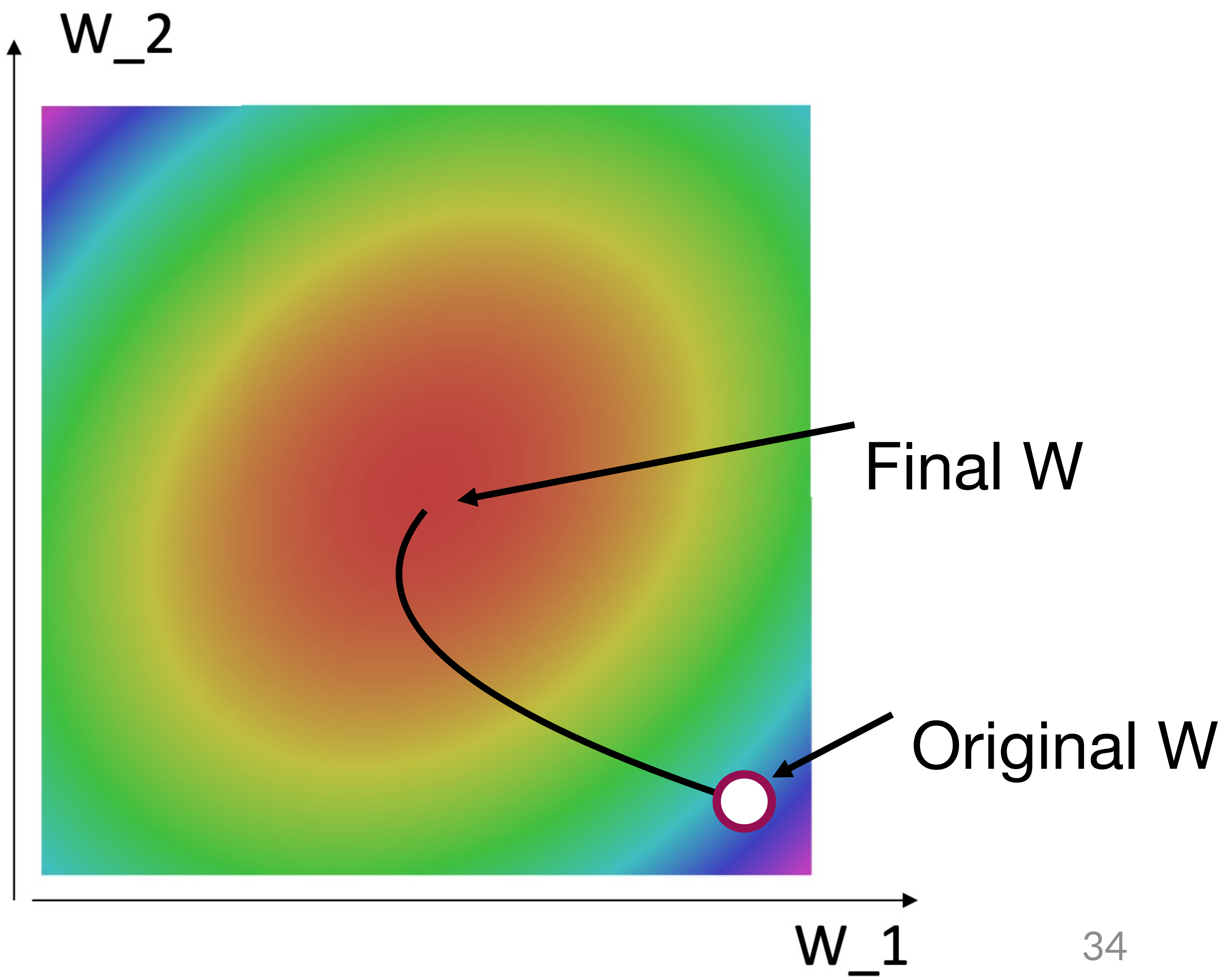
Gradient Descent

- 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)

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

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

```python
# 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)

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

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

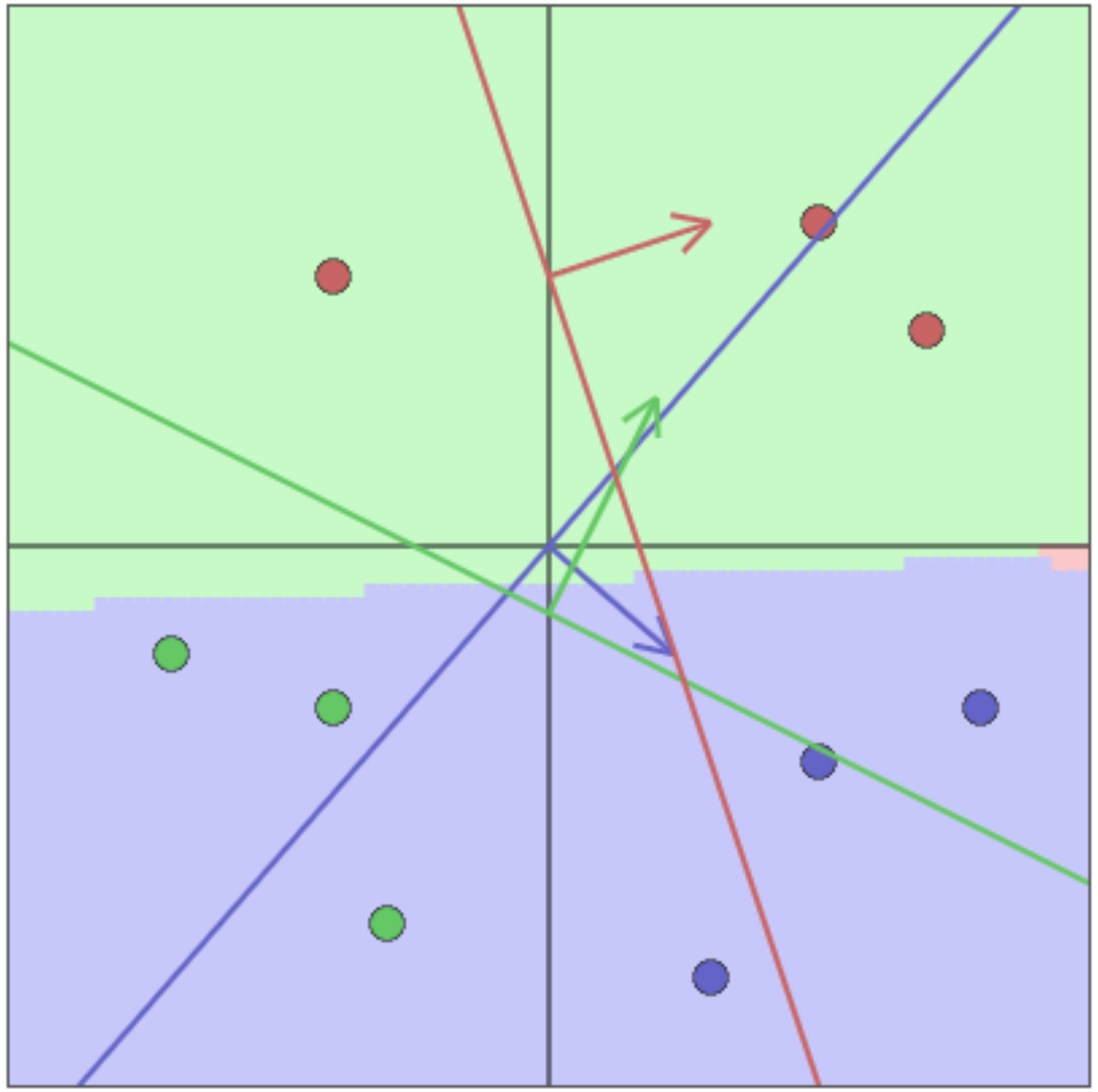
Think of loss as an expectation over the full **data distribution** p_{data}

Approximate expectation via sampling

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

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

Interactive Web Demo



W[0,0]	W[0,1]	b[0]
2.30 / -0.17	2.00 / 0.14	0.00 / 0.33

W[1,0]	W[1,1]	b[1]
2.00 / 0.42	-4.00 / -0.61	0.50 / -0.22

W[2,0]	W[2,1]	b[2]
3.00 / 0.12	-1.00 / 0.32	-0.50 / -0.11

Step size: 0.09976

Single parameter update

Start repeated update

Stop repeated update

Randomize parameters

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	1.95	-0.10	0.60	0.00
0.80	0.30	0	2.44	0.90	1.60	0.16
0.30	0.80	0	2.29	-2.10	-0.40	0.00
-0.40	0.30	1	-0.32	-1.50	-2.00	2.68
-0.30	0.70	1	0.71	-2.90	-2.10	6.41
-0.70	0.20	1	-1.21	-1.70	-2.80	1.49
0.70	-0.40	2	0.81	3.50	2.00	2.50
0.50	-0.60	2	-0.05	3.90	1.60	3.30
-0.40	-0.50	2	-1.92	1.70	-1.20	4.18
						mean:
						2.30

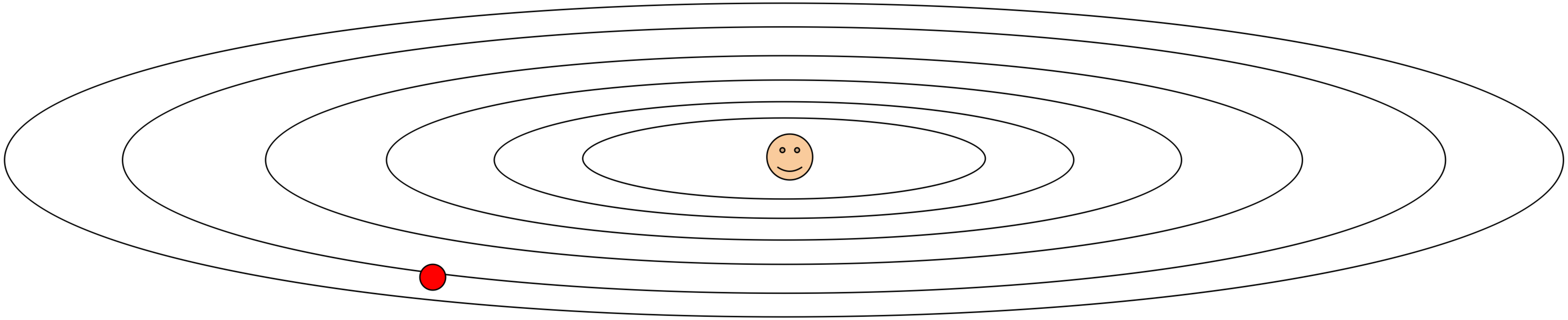
Total data loss: 2.30
Regularization loss: 3.93
Total loss: 6.23

L2 Regularization strength: 0.10000

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

Problems with SGD

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



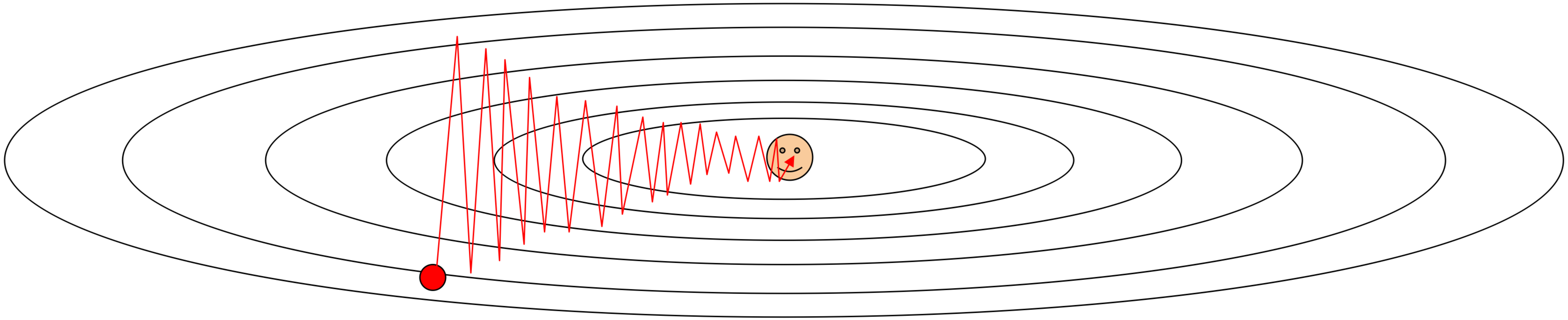
Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with SGD

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



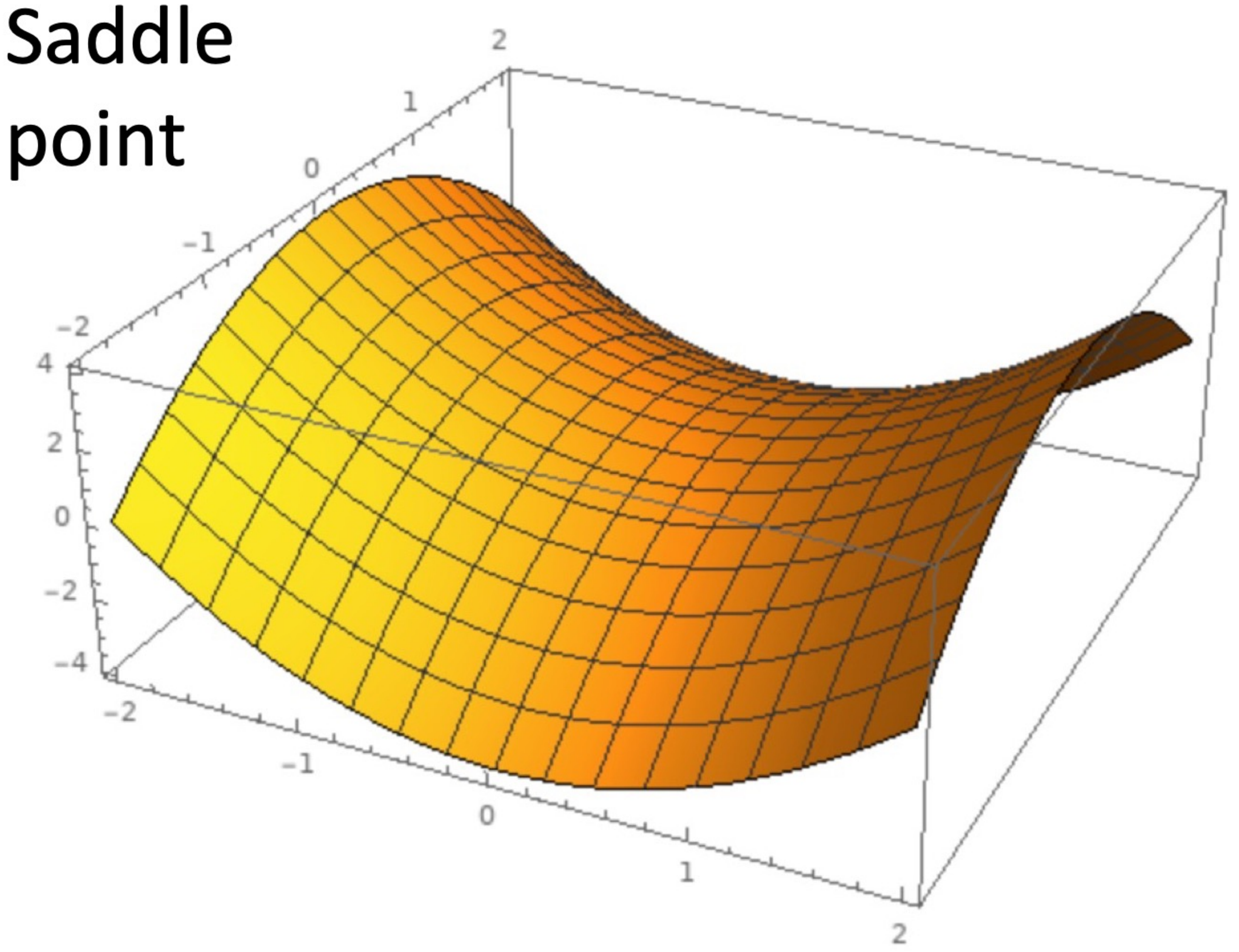
Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with SGD

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

Local
Minimum

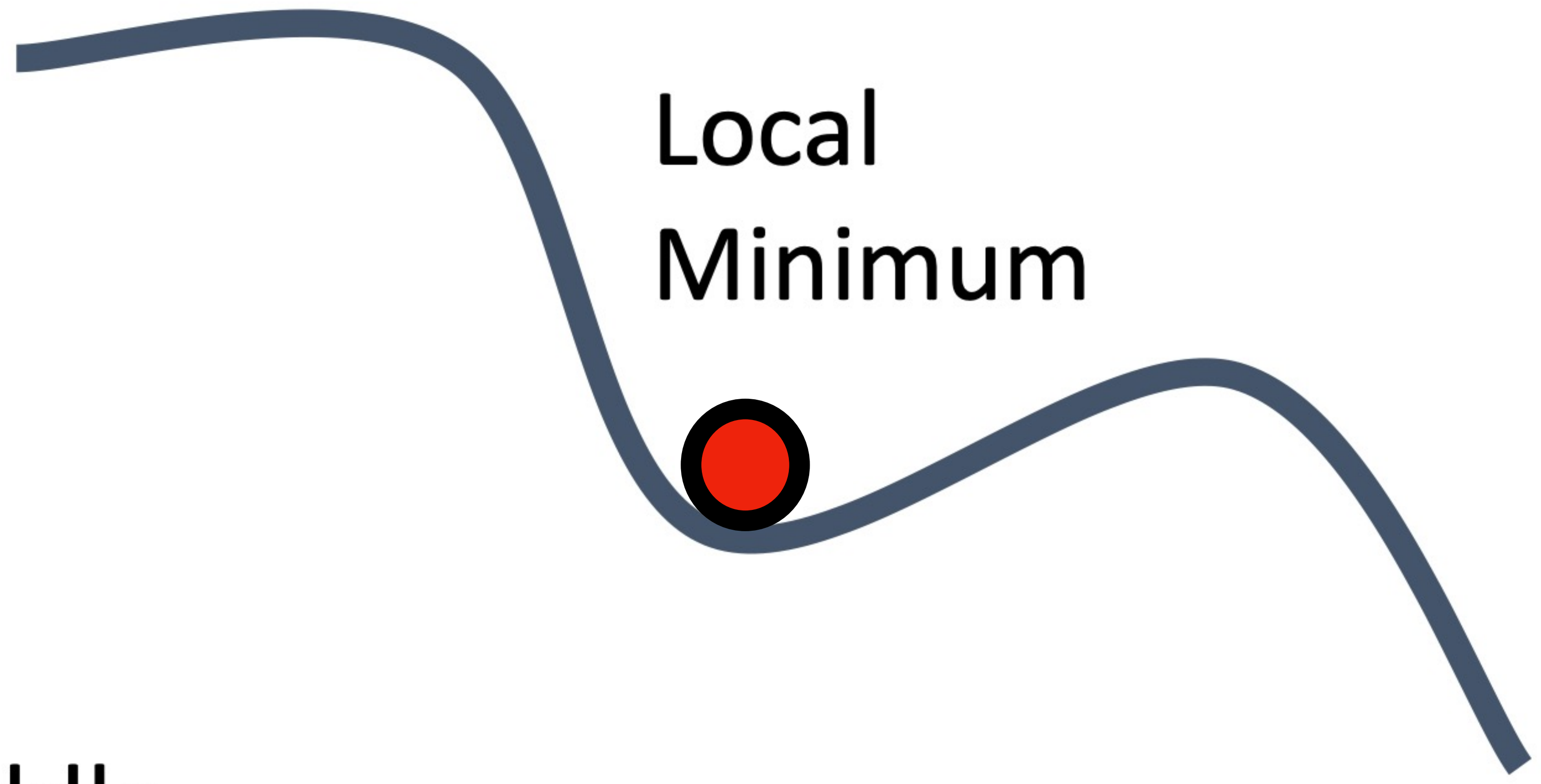
Saddle
point

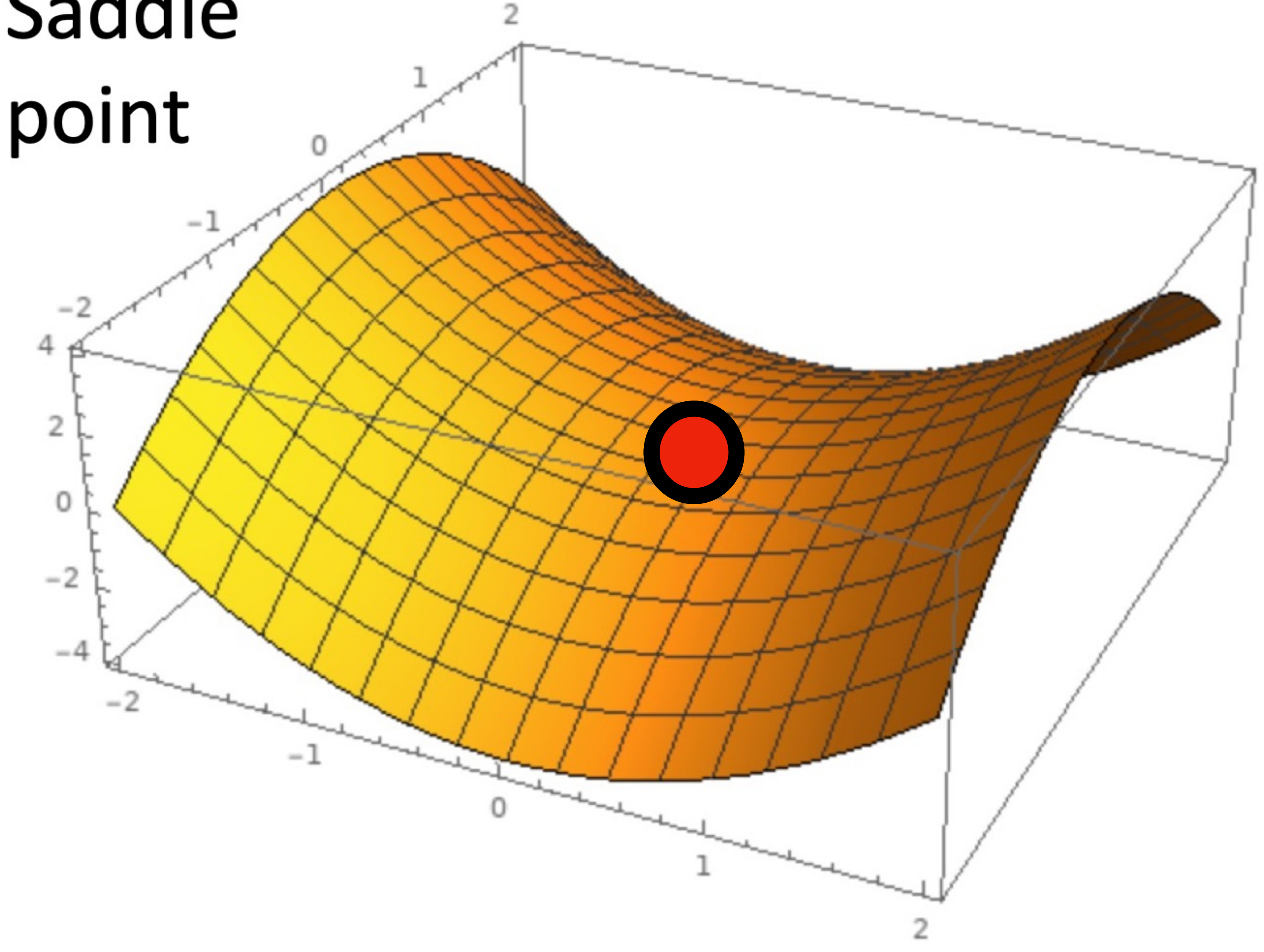


Problems with SGD

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

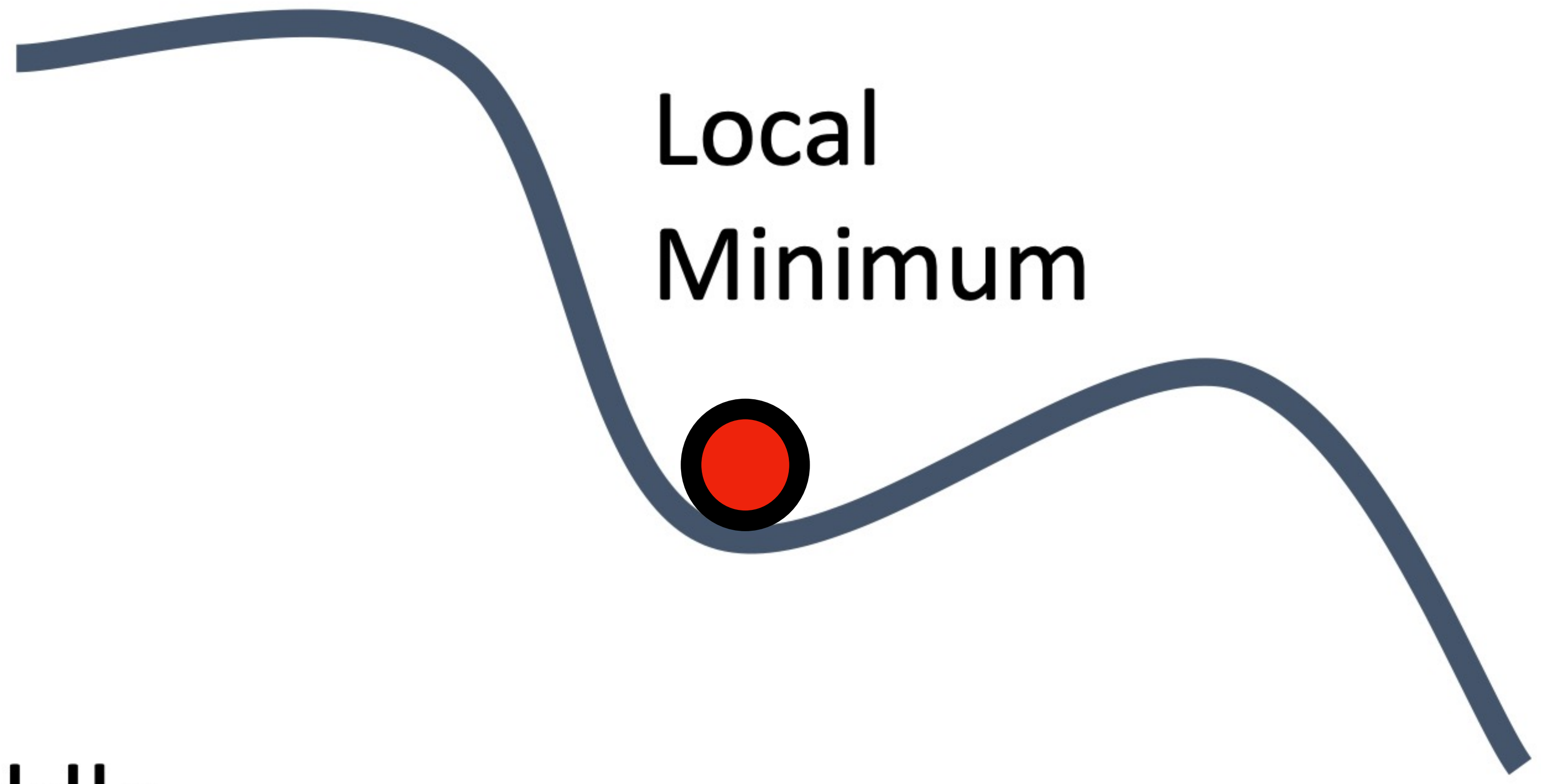
Zero gradient, gradient descent gets stuck

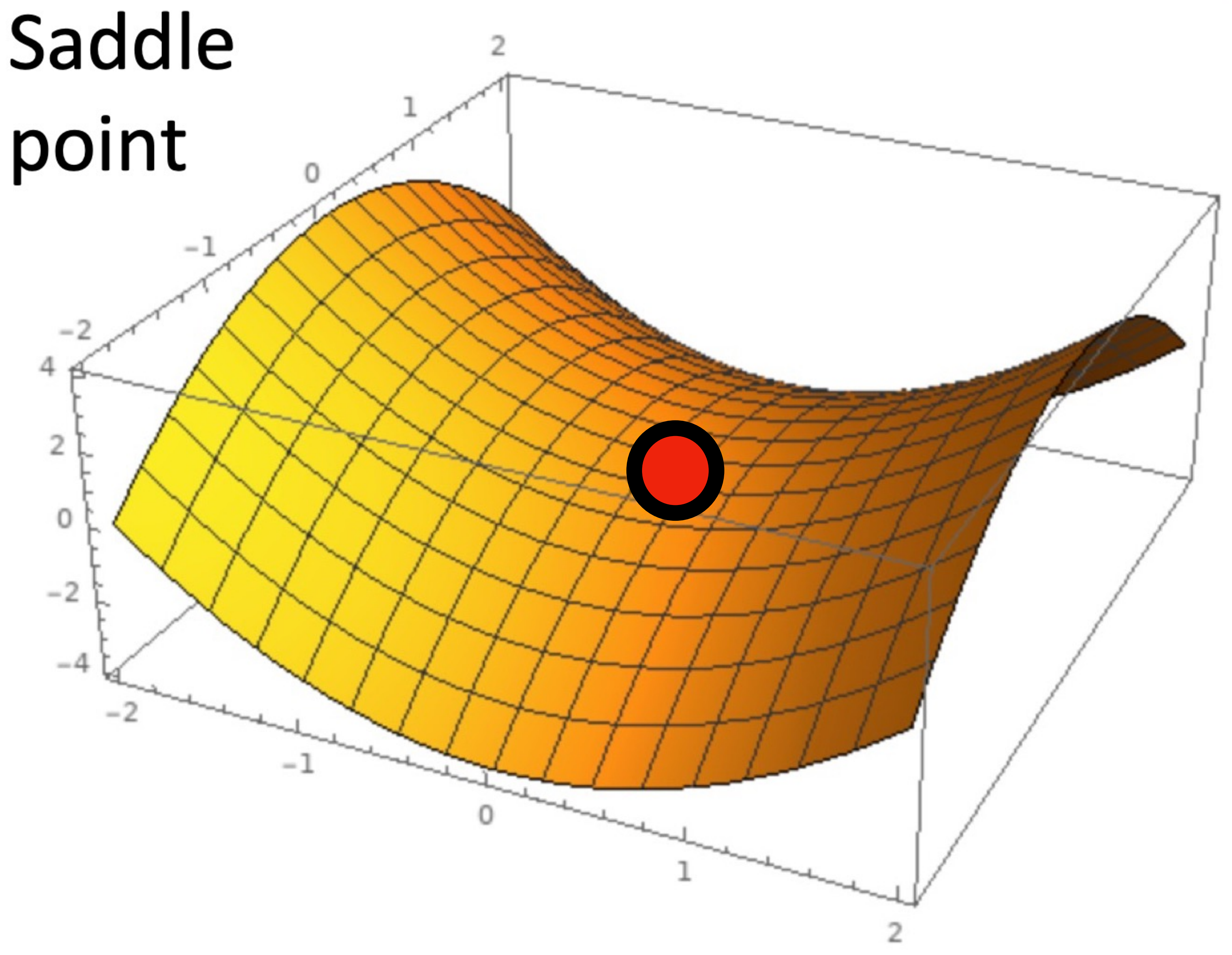




Problems with SGD

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

SGD + Momentum

SGD

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

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

SGD + Momentum

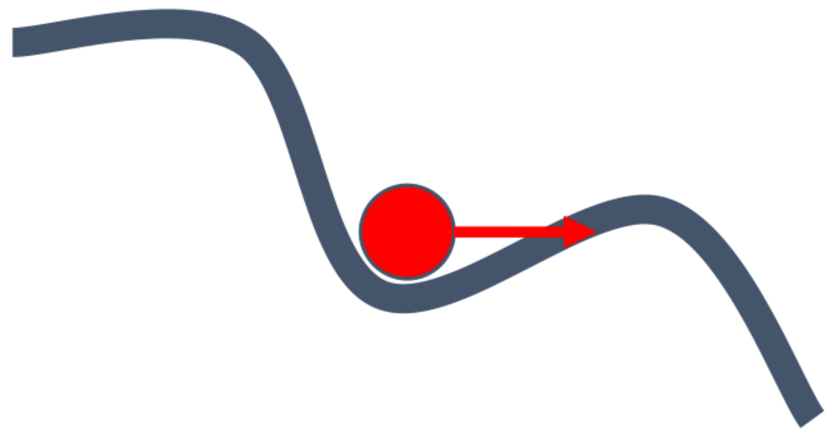
$$v_{t+1} = \rho v_t + \nabla L(w_t)$$
$$w_{t+1} = w_t - \alpha v_{t+1}$$

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

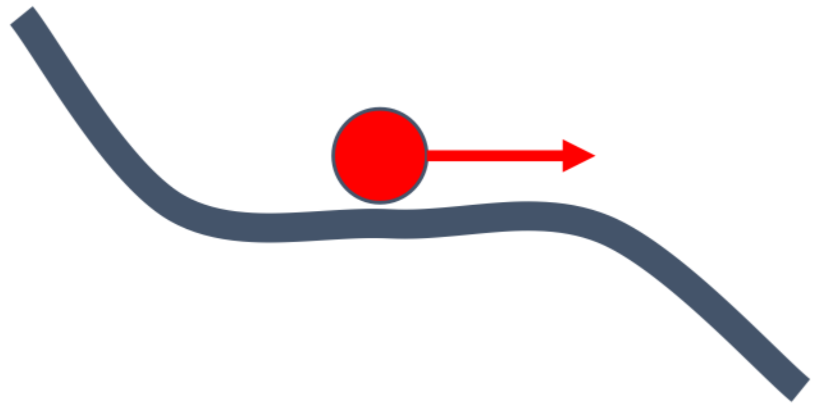
- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho = 0.9 or 0.99

SGD + Momentum

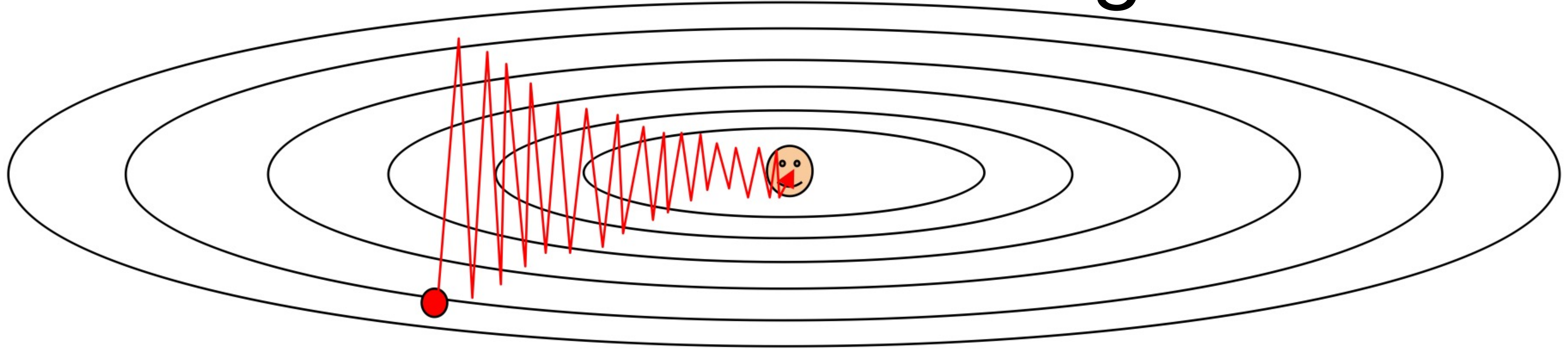
Local Minima



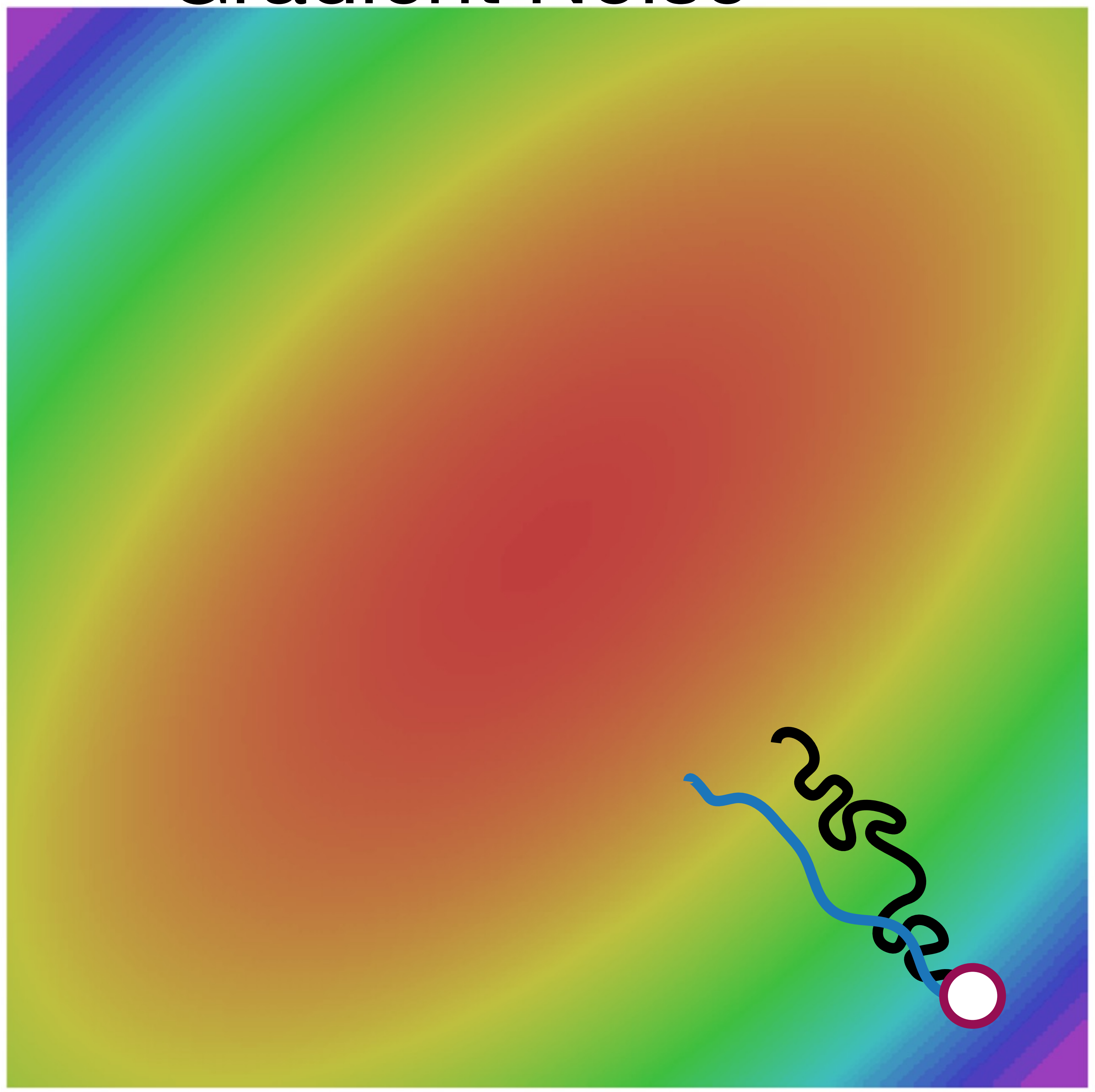
Saddle Points



Poor Conditioning



Gradient Noise

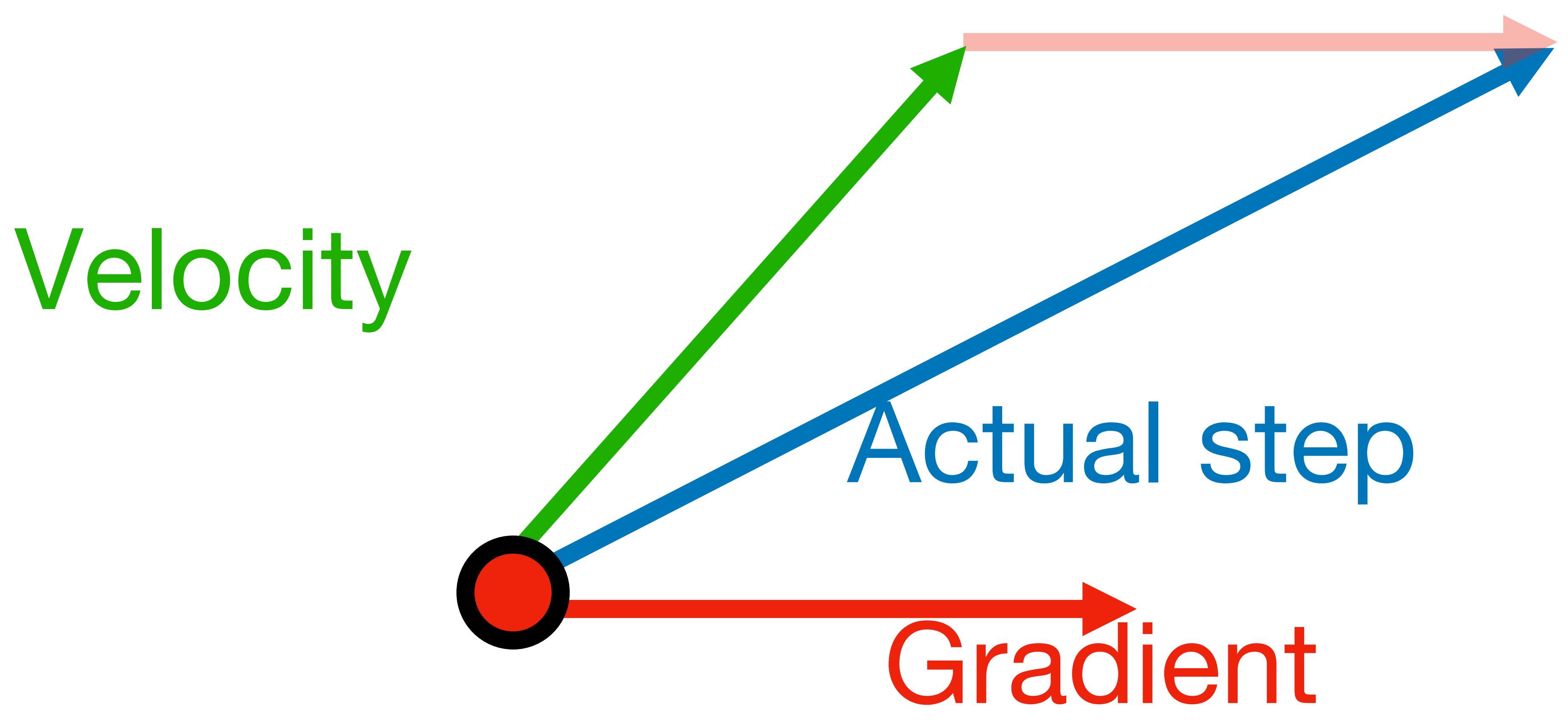


SGD SGD+Momentum

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

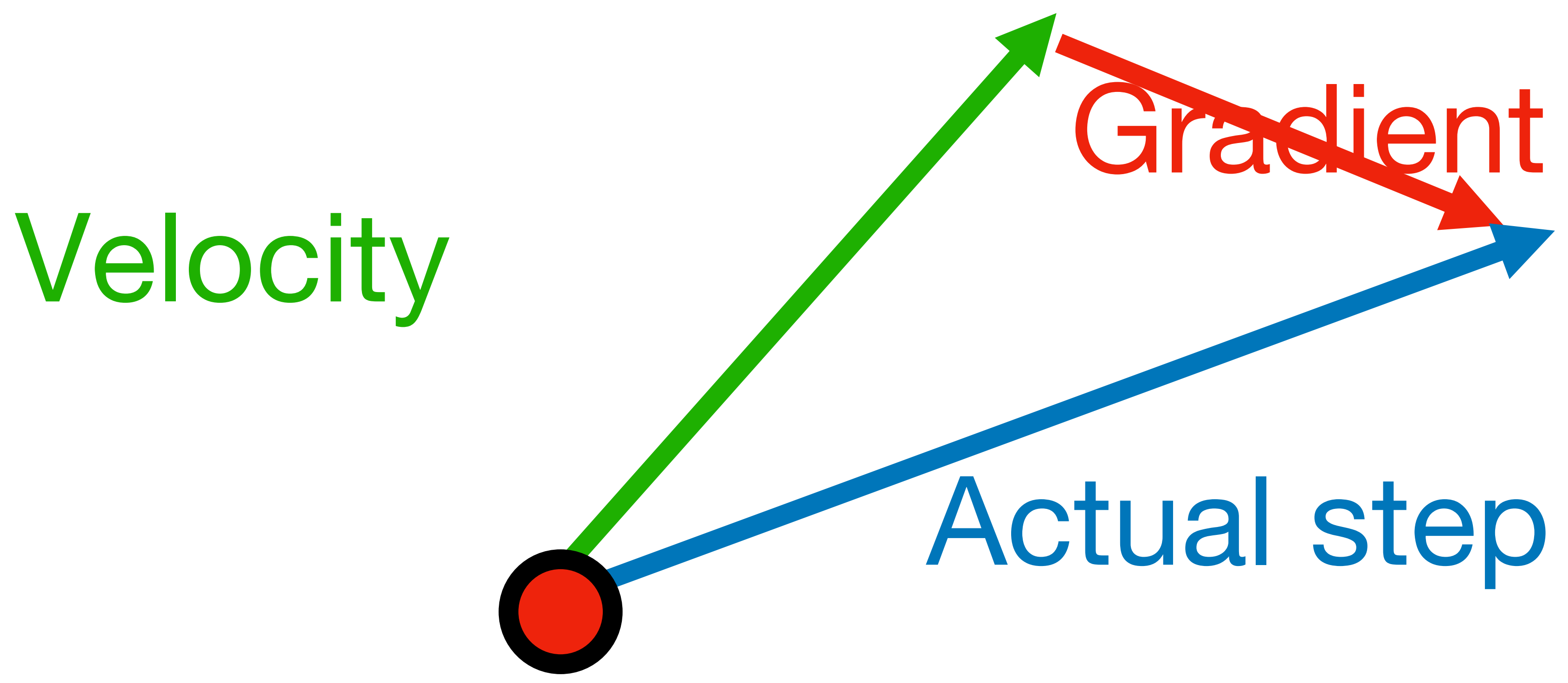
SGD + Momentum

Momentum update:



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

Nesterov 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

DeepRob

Lecture 3

Regularization + Optimization

University of Michigan | Department of Robotics