CMPU 366 · Natural Language Processing

Gradient Descent

24 September 2025

Where are we?

Logistic regression is a supervised classification method:

From a set of labeled documents, we learn a classifier that can predict the label for new documents.

There can be any number of possible labels, but the easiest – and most common – case is binary classification.

1. Learn weights

1. Learn weights

1. Learn weights

2. Predict probability

via the sigmoid (logistic) function, whose input is the dot product between the weight vector and the feature vector

$$P(y = 1 \mid x) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

$$= \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

1. Learn weights

2. Predict probability

via the sigmoid (logistic) function, whose input is the dot product between the weight vector and the feature vector

3. Predict label

via a threshold on the predicted probability

$$P(y = 1 \mid x) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

$$= \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

$$\text{decision}(\mathbf{x}) = \begin{cases} 1 & \text{if } P(y = 1 \mid \mathbf{x}) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

1. Learn weights
via
a loss function
and
an optimization algorithm

2. Predict probability
via the sigmoid (logistic) function, whose input is the dot product between the weight vector and the feature vector

$$P(y = 1 \mid x) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$
$$= \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

3. Predict label
via a threshold on the predicted probability

$$\text{decision}(\mathbf{x}) = \begin{cases} 1 & \text{if } P(y = 1 \mid \mathbf{x}) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

This is the classifier's guess of y, so we can call it $\hat{y}$.

$$\boxed{P(y = 1 \mid x)} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

$$= \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

Learning weights

We know the correct label y (either 0 or 1) for each $\mathbf{x}$ in the training data, but what the system produces is an estimate, $\hat{y}$ (between 0 and 1).

We want to set $\mathbf{w}$ and b to minimize the distance between our estimate $\hat{y}$ and the true y, which we denote call the *loss*, $L(\hat{y}, y)$.

Goal: Maximize the probability of the correct label

$$P(y \mid x) = \hat{y}^{y} \, (1 - \hat{y})^{1-y}$$

Goal: Maximize the probability of the correct label

$$P(y \mid x) = \boxed{\hat{y}^{y}}(1 - \hat{y})^{1-y}$$

This part matters when y = 1

Goal: Maximize the probability of the correct label

$$P(y \mid x) = \boxed{\hat{y}^{y}}\,\boxed{(1 - \hat{y})^{1-y}}$$

This part matters when y = 1

This part matters when y = 0

Goal: Maximize the probability of the correct label

$$P(y \mid x) = \hat{y}^{y} \, (1 - \hat{y})^{1-y}$$

$$\log P(y \mid x) = \log \left[\hat{y}^{y} \, (1 - \hat{y})^{1-y}\right]$$

$$= y \log \hat{y} + (1 - y) \log (1 - \hat{y})$$

Goal: Maximize the probability of the correct label

Goal: Minimize the cross-entropy loss (negative log likelihood)

$$-\log P(y \mid x) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

$$L_{\mathrm{CE}}(\hat{y}, y) = -[y \log \sigma(\mathbf{wx} + b) + (1 - y) \log (1 - \sigma(\mathbf{wx} + b))]$$

Today we'll consider how we minimize this loss to learn the optimal weights – and then we'll revisit the important question of evaluating a classifier.

Stochastic gradient descent

Our goal: minimize the loss

Let's make explicit that the loss function is parameterized by weights $\theta = (\mathbf{w}, b)$.

And we'll represent $\hat{y}$ as $f(x; \theta)$ to make the dependence on θ more obvious.

We want the weights that minimize the loss, averaged over all examples:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

Model's guess

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(\boxed{f(x^{(i)}; \theta)}, \boxed{y^{(i)}})$$

Model's guess

Actual label

the loss for one training example

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} \boxed{L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})}$$

m = length of dataset

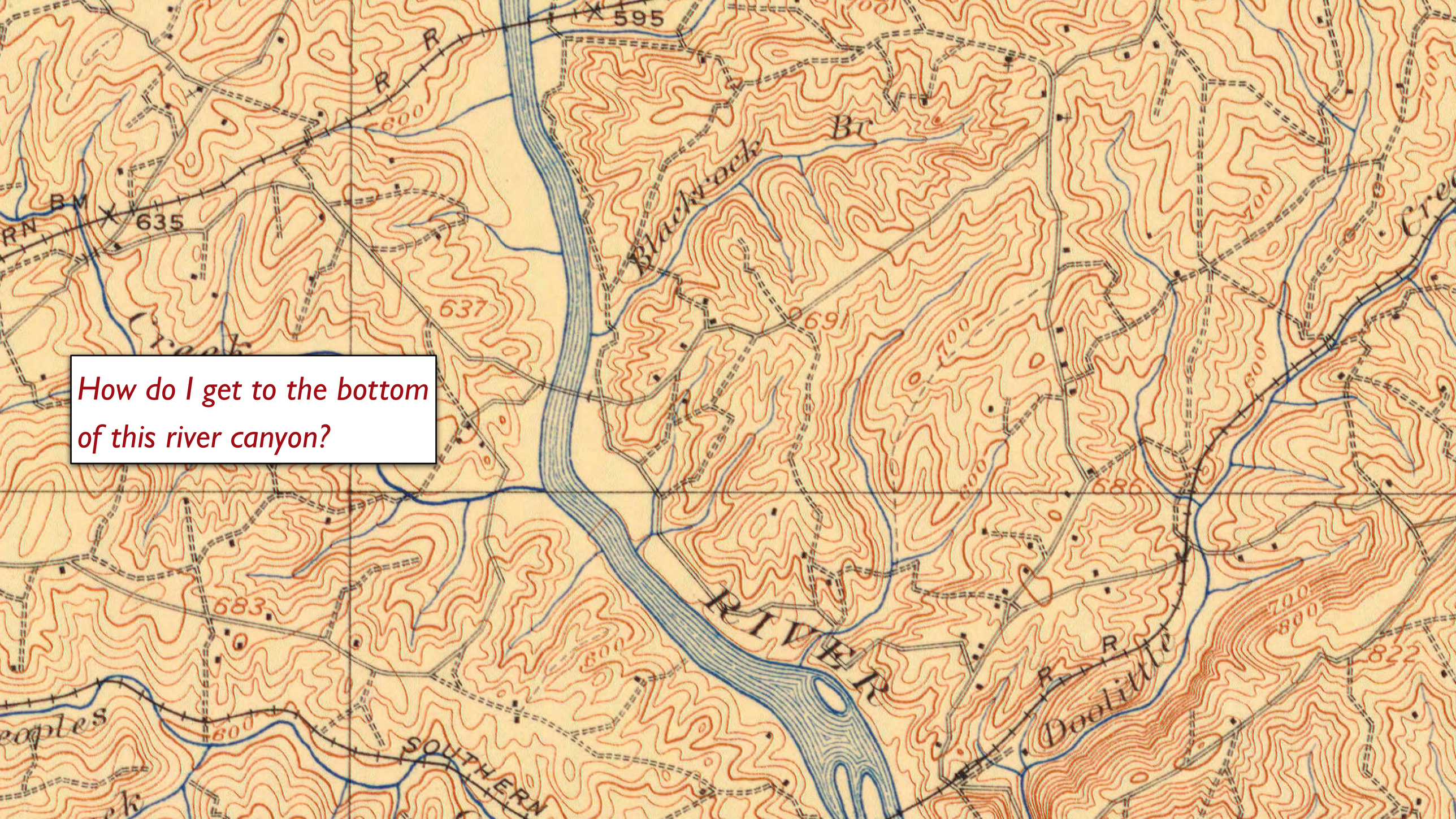
$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{\boxed{m}} \sum_{i=1}^{\boxed{m}} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

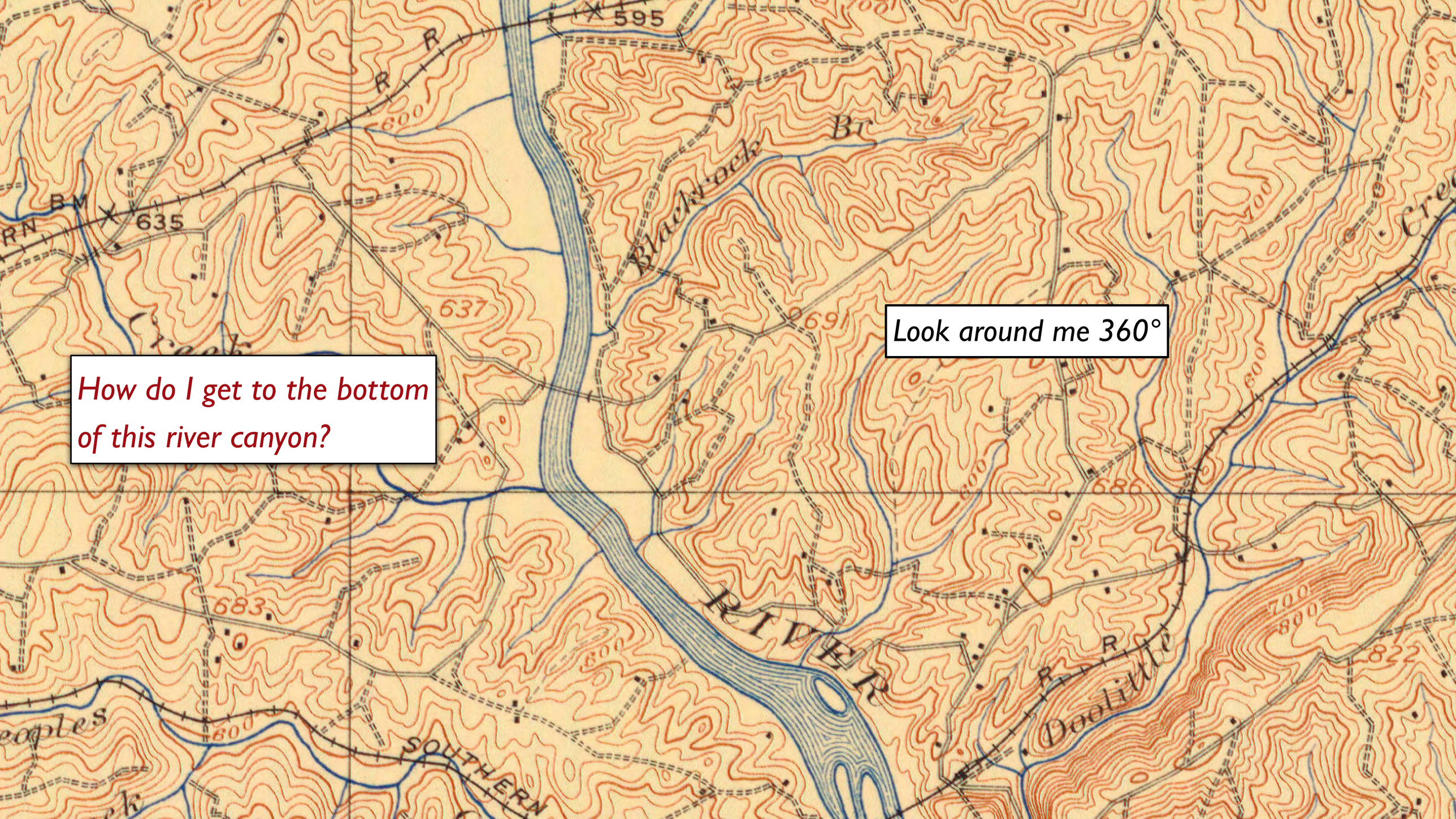
the average loss over all training examples

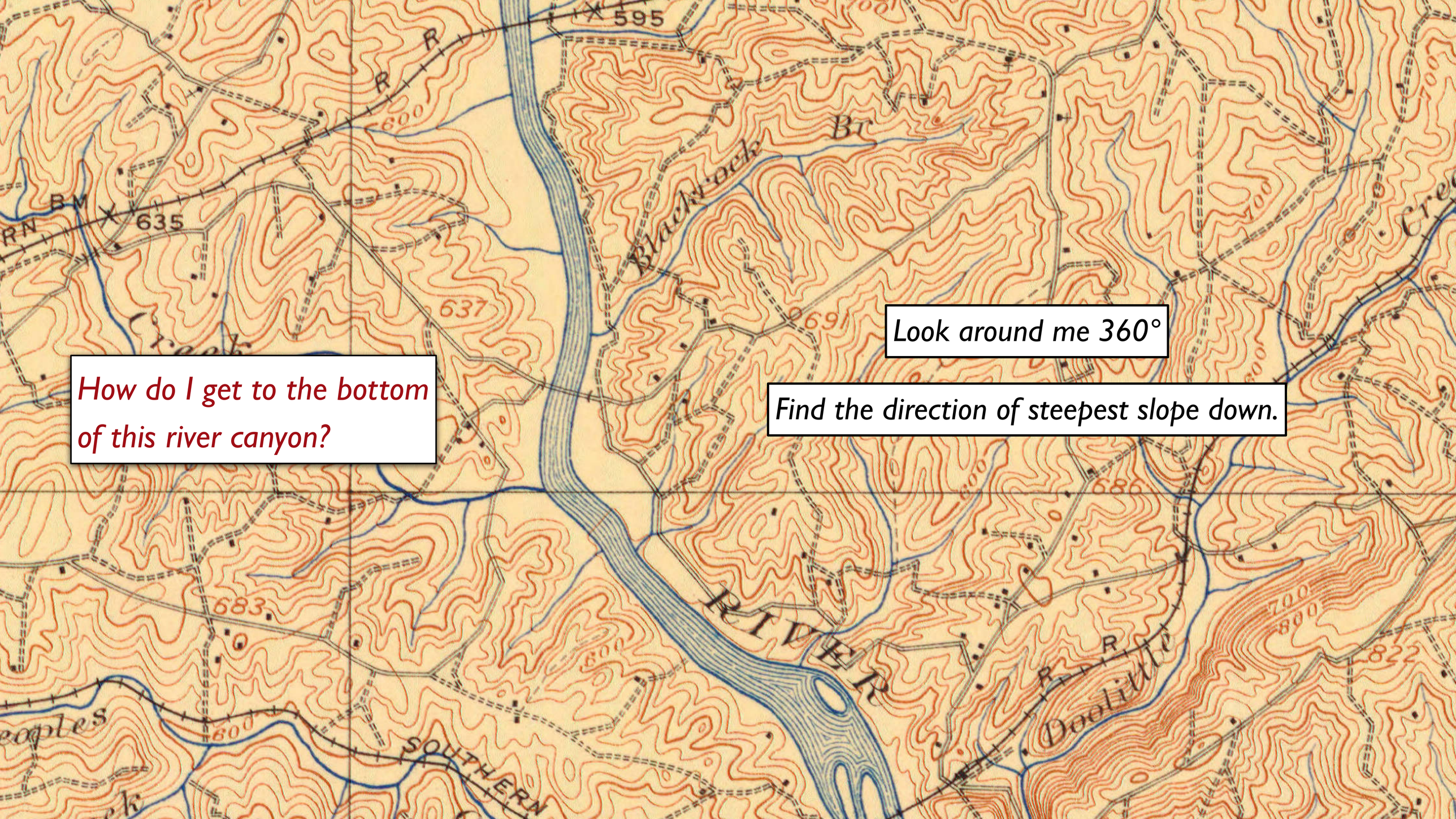
$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \boxed{\frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})}$$

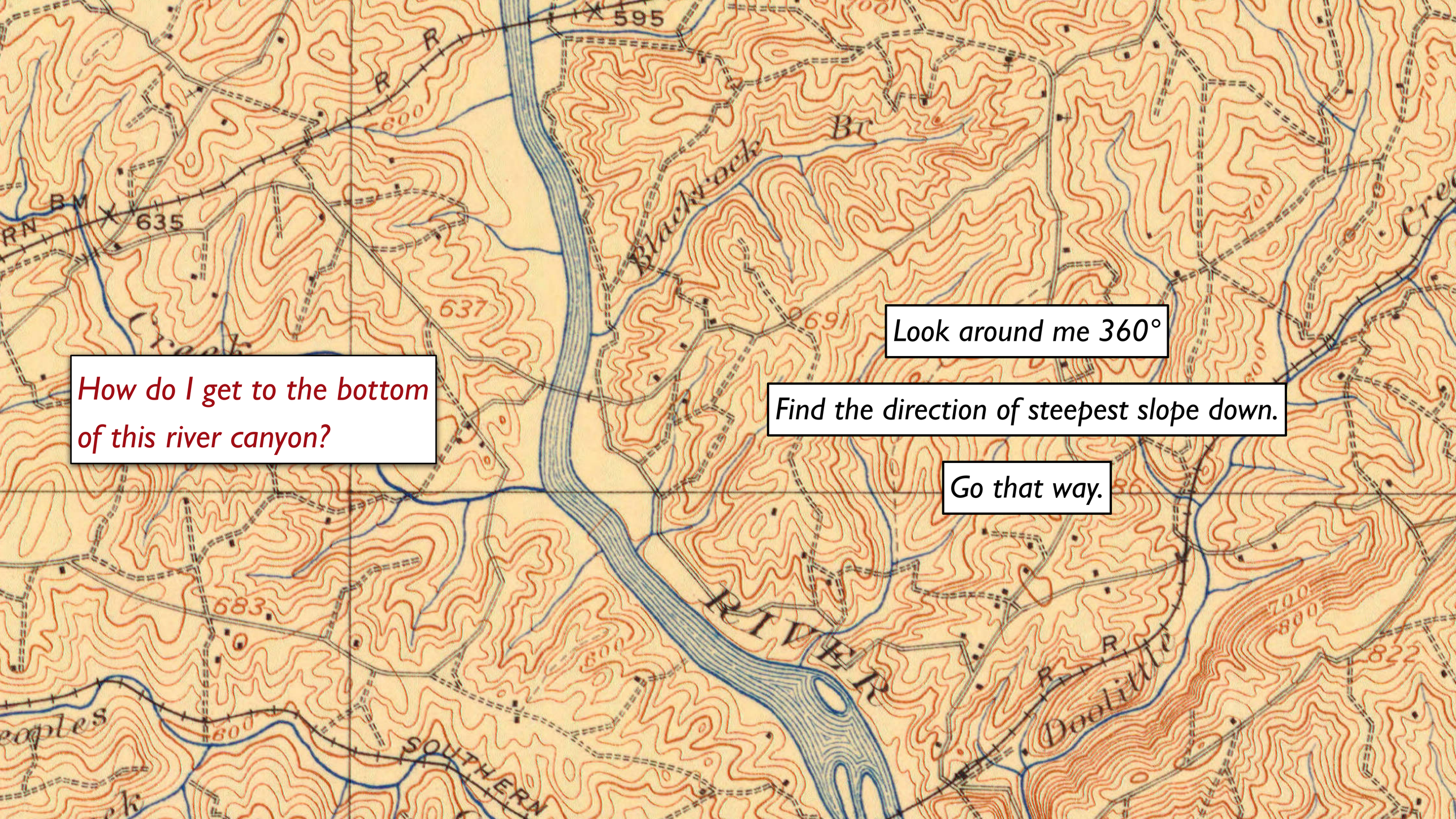
*the weights θ (**w** and b) that have the lowest average loss*

$$\hat{\theta} = \boxed{\underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})}$$


How do I get to the bottom of this river canyon?
595
635
637
691
683
686
822
600
700
800
Black rock
Br
RIVER
Doolittle
SOUTHERN
RN
BM

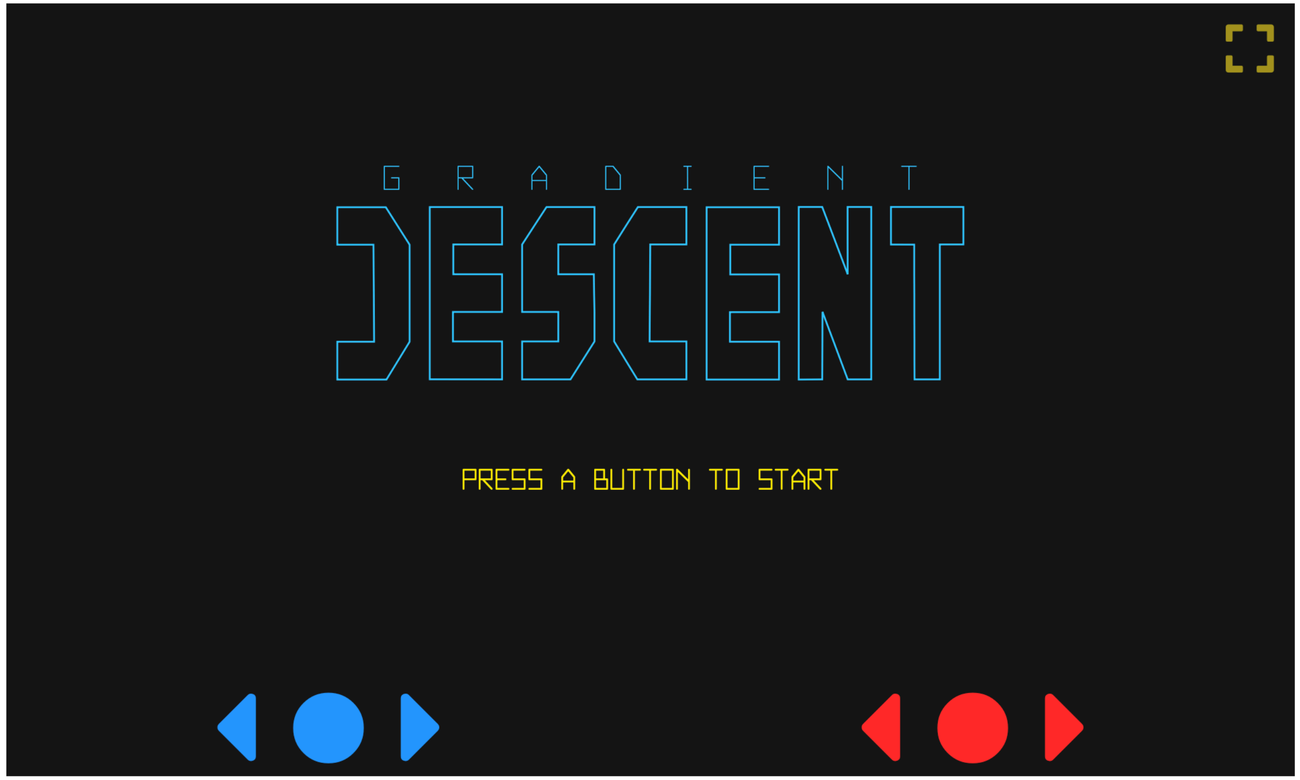

How do I get to the bottom of this river canyon?
Look around me 360°
595
635
637
691
686
683
822
700
800
600
Black rock
Br
RIVER
Doolittle
SOUTHERN
R
BM
RN


595
635
637
691
683
686
822
700
800
600
Blackrock
Br
RIVER
Doolittle
SOUTHERN
Peoples
Creek
How do I get to the bottom of this river canyon?
Look around me 360°
Find the direction of steepest slope down.


How do I get to the bottom of this river canyon?
Look around me 360°
Find the direction of steepest slope down.
Go that way.
595
635
637
691
683
600
700
800
822
Black rock
Br
RIVER
Doolittle
SOUTHERN
BM
RN

www.i-am.ai/gradient-descent.html

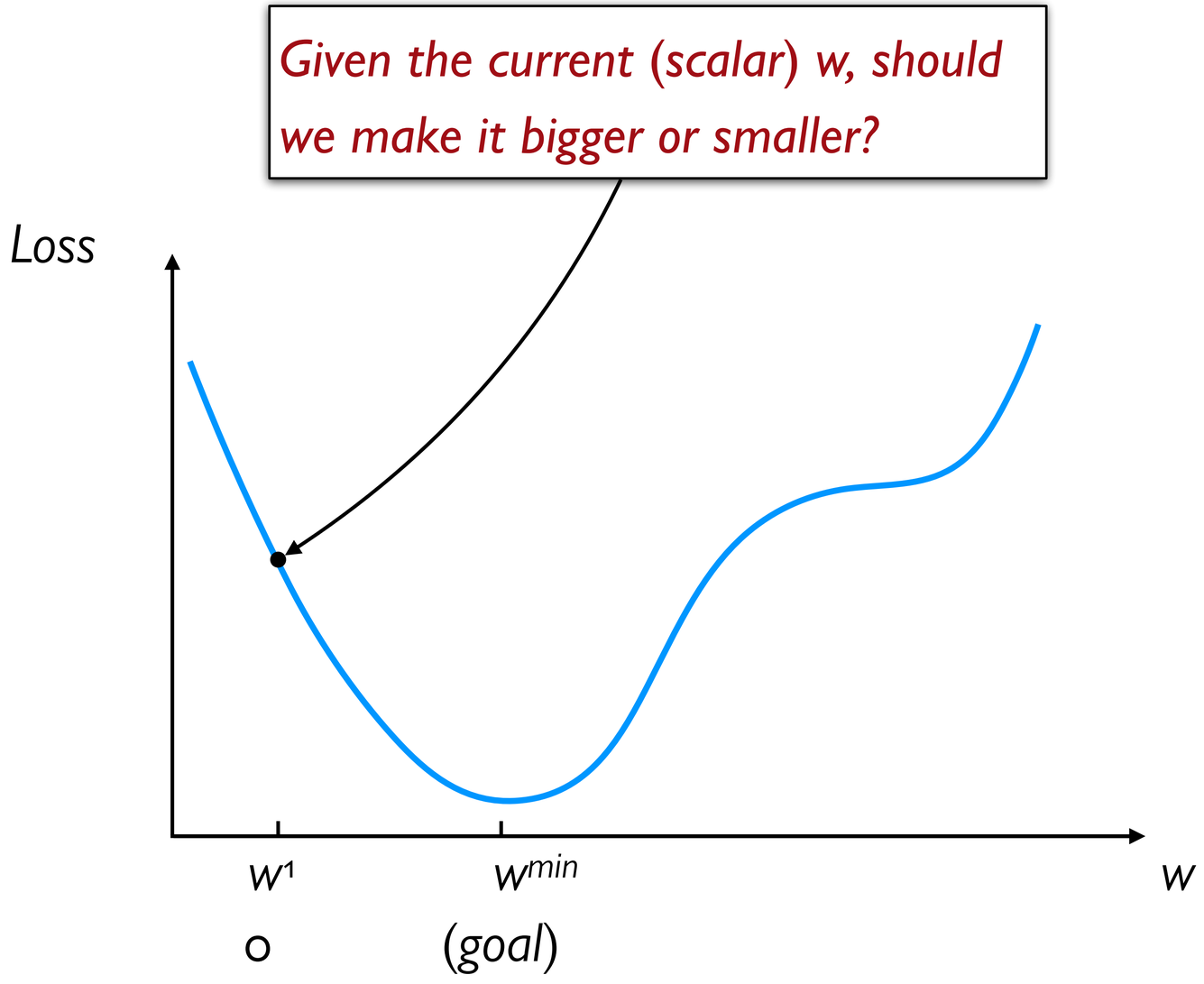
to win.



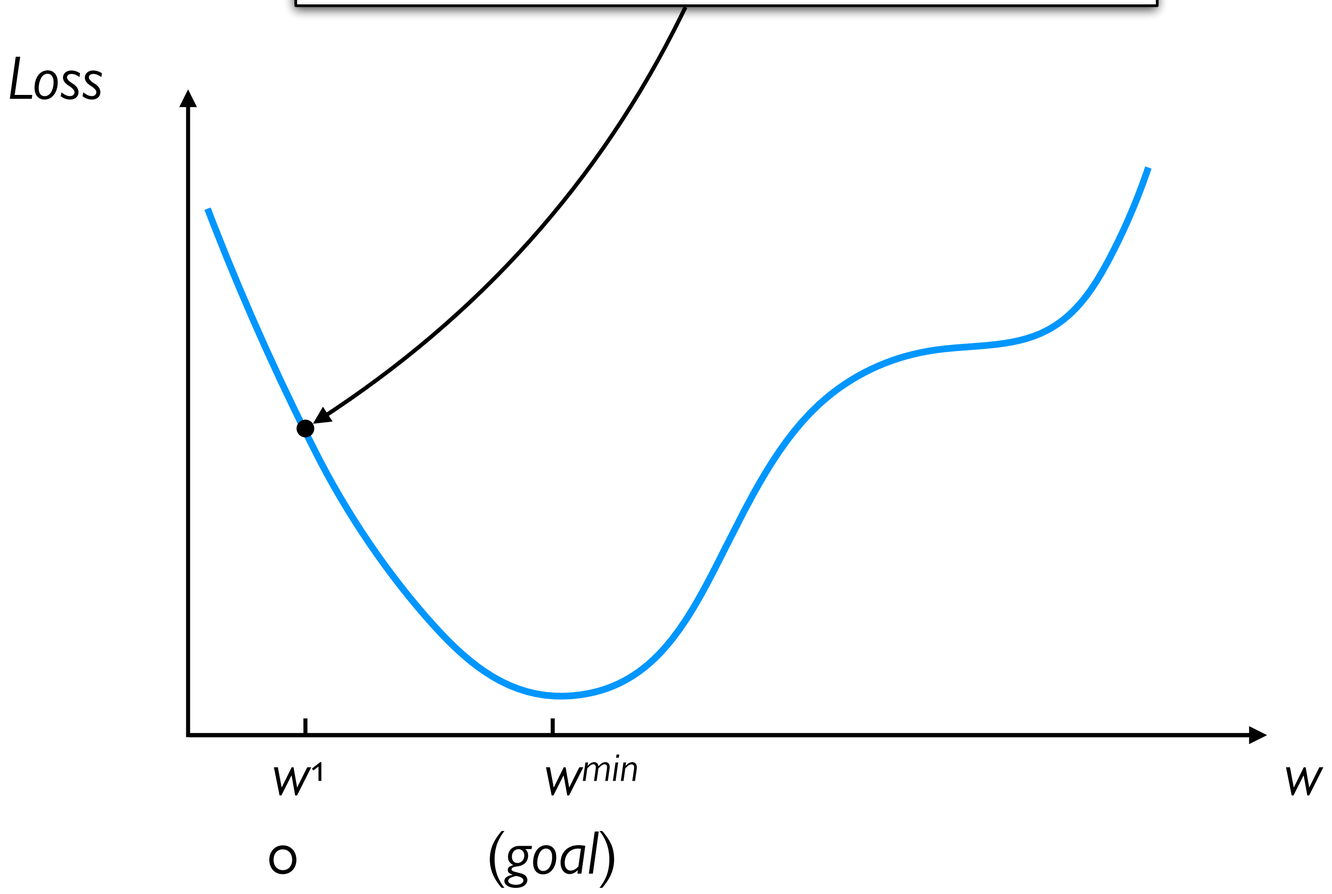
After playing a couple of times you probably figured out that the quickest way to find the deepest point in the ocean floor is by paying attention to the slope found by the probe: how steep the floor was at that point and in which direction it was inclined. Although you can't see the bottom and don't have a full picture of

i-am.ai/gradient-descent.html

Gradient descent is designed for vectors – like the weights we're learning – but it's easier to think about the simpler case of a scalar.

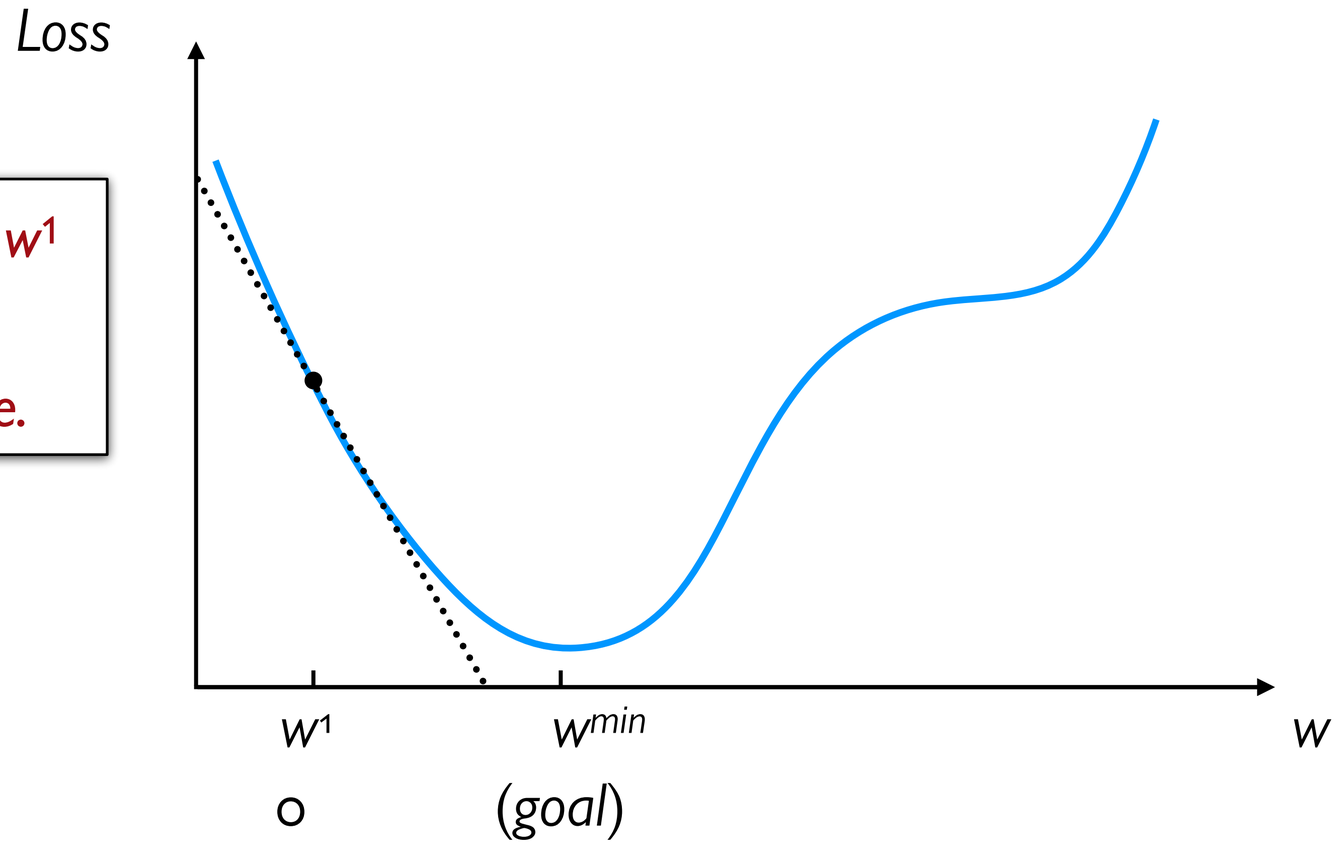

Given the current (scalar) w, should we make it bigger or smaller?
Loss
w
w^1
w^{min}
(goal)

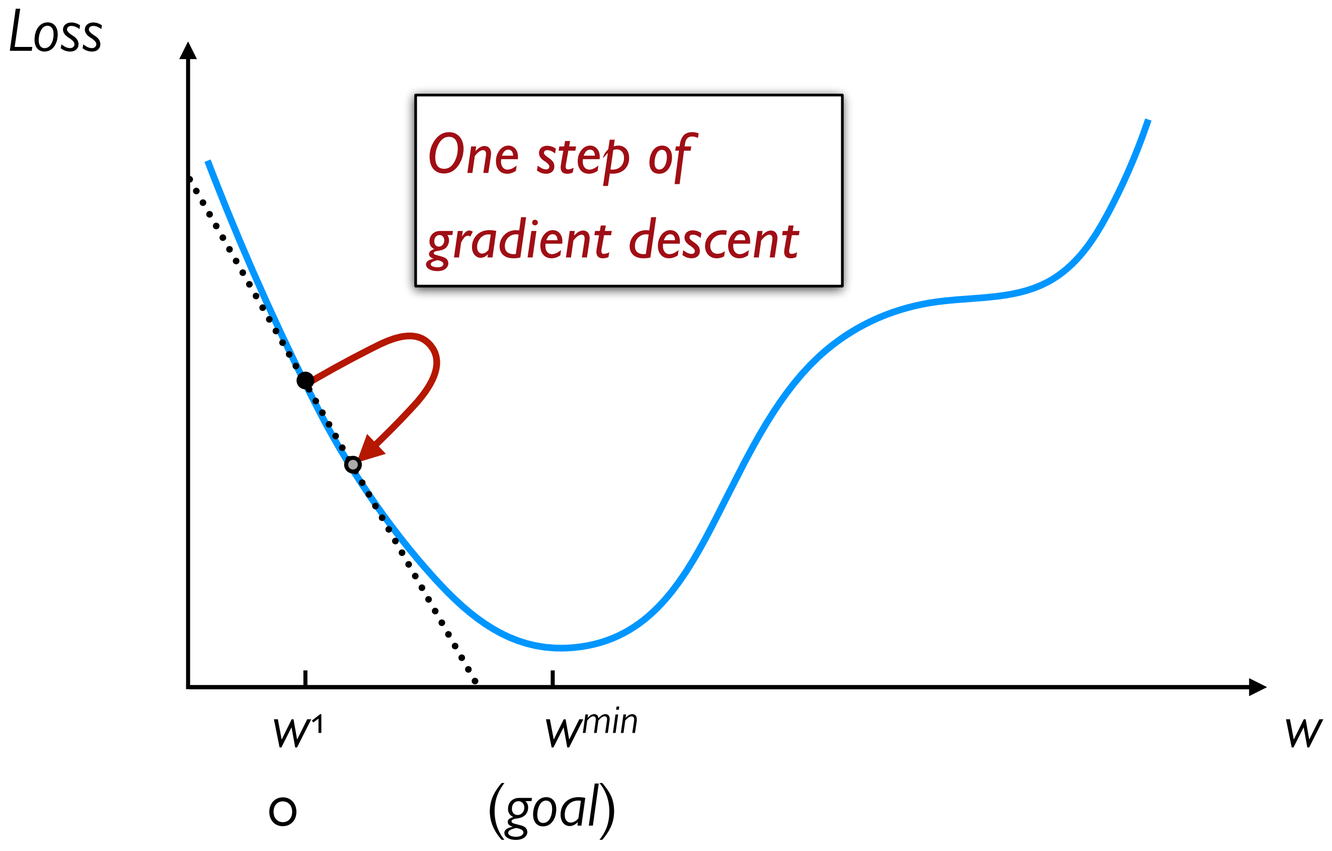
Given the current (scalar) w, should we make it bigger or smaller?

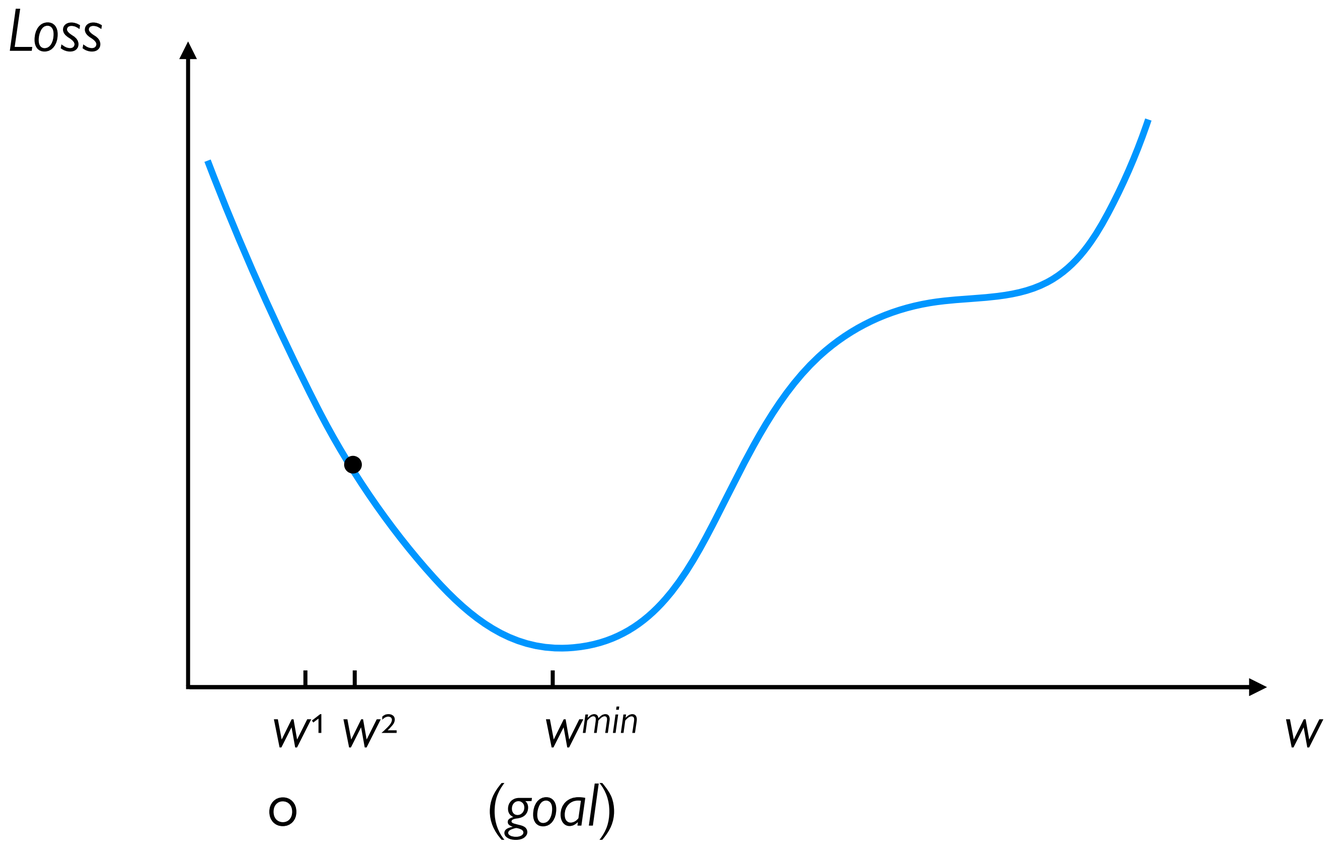


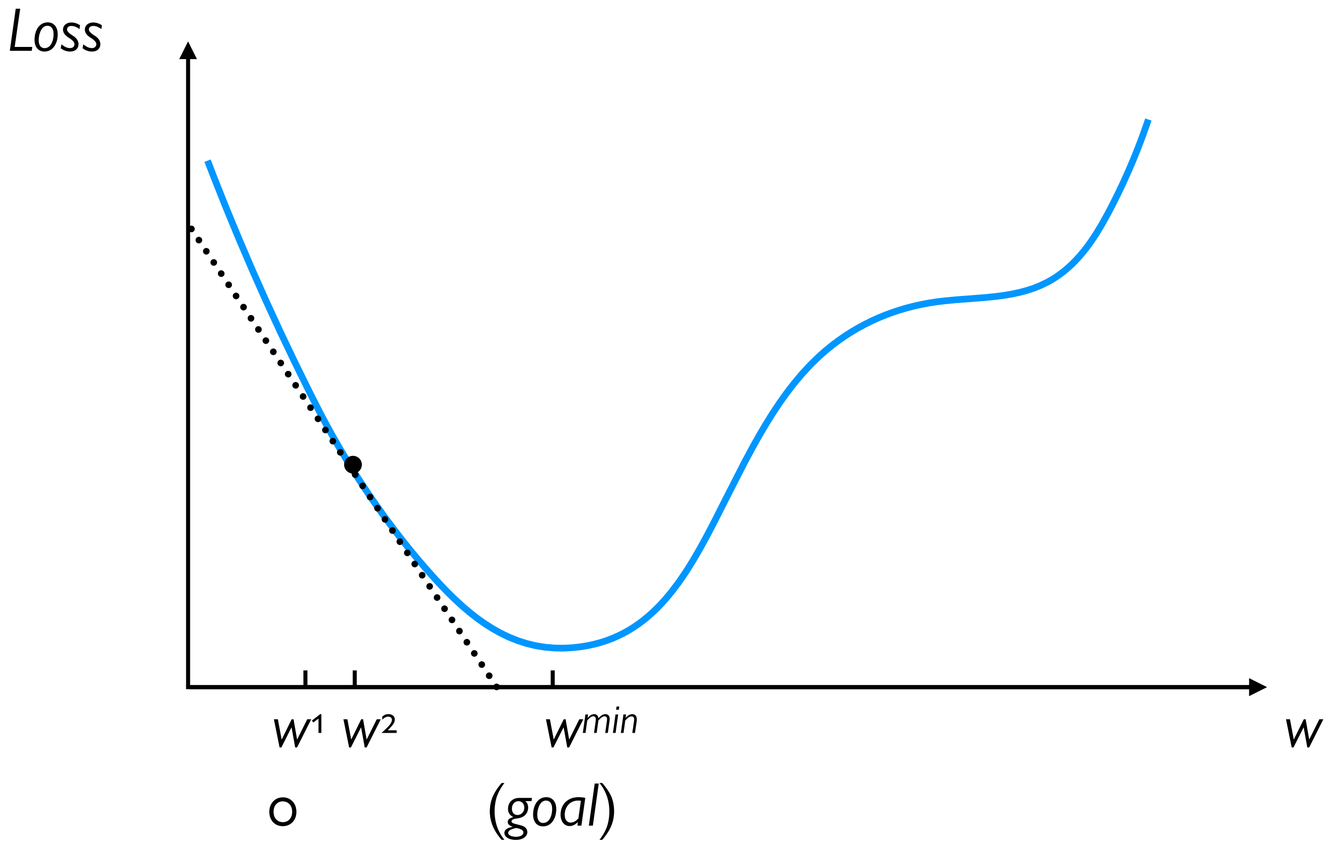
Move w in the reverse direction from the slope of the function.

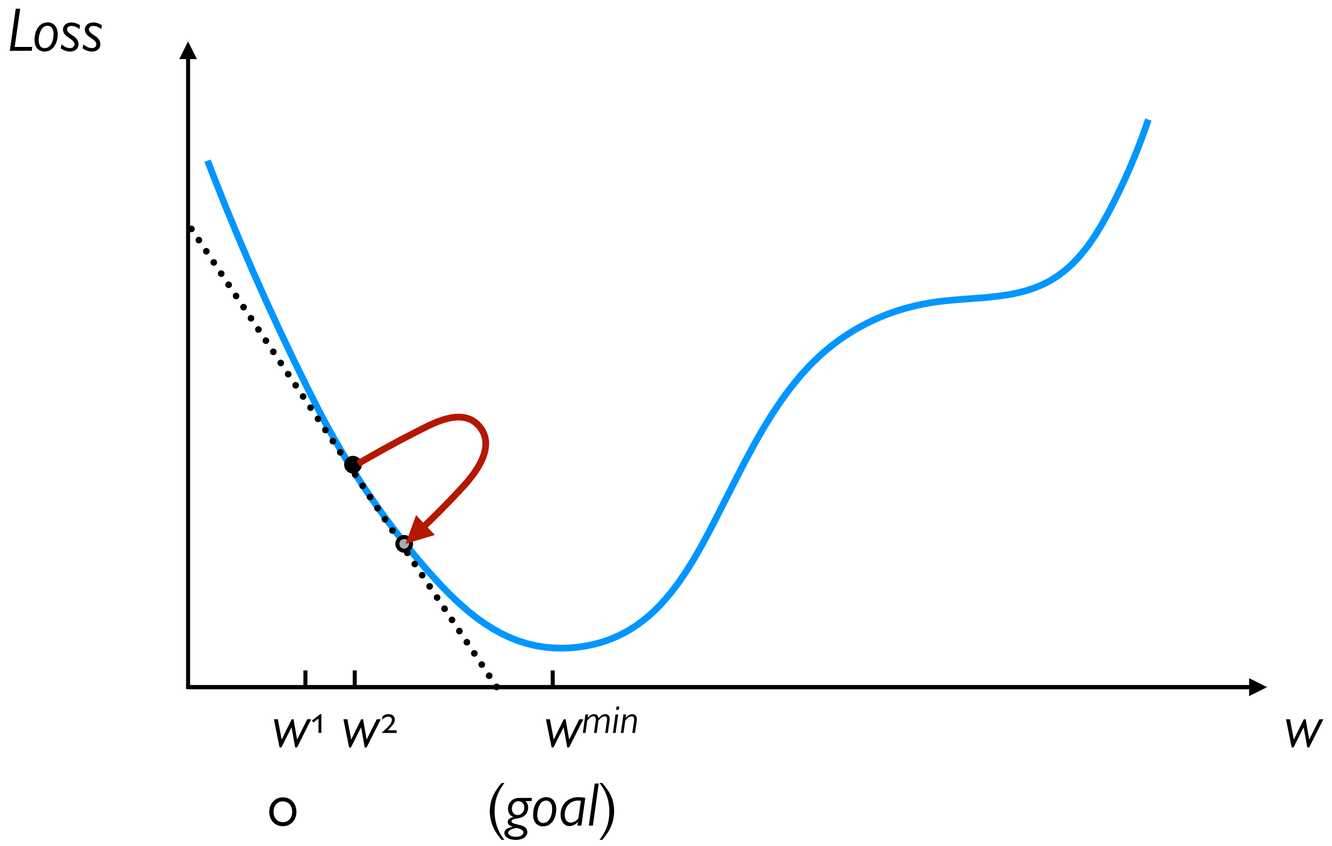
The slope of loss at w^1 is negative, so we should move positive.

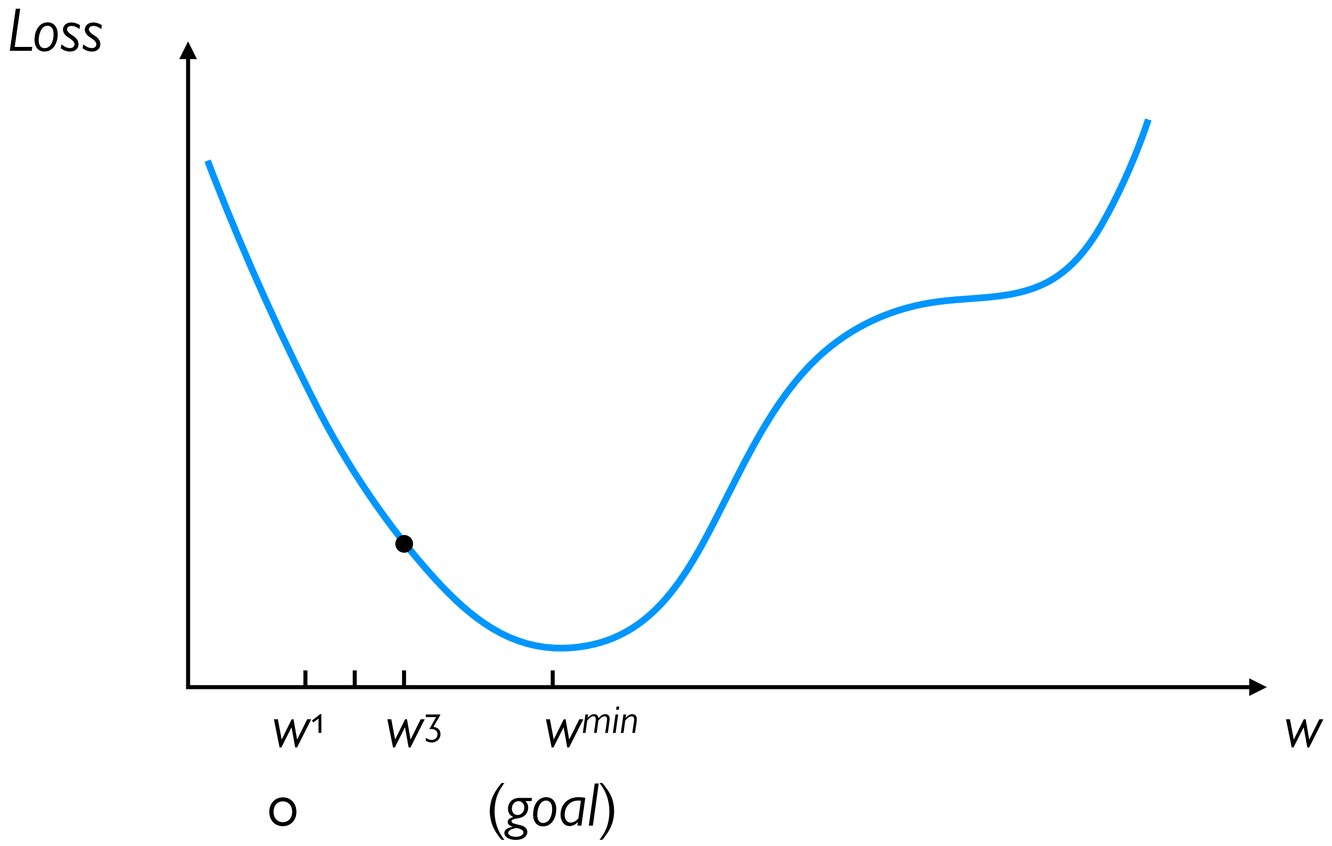


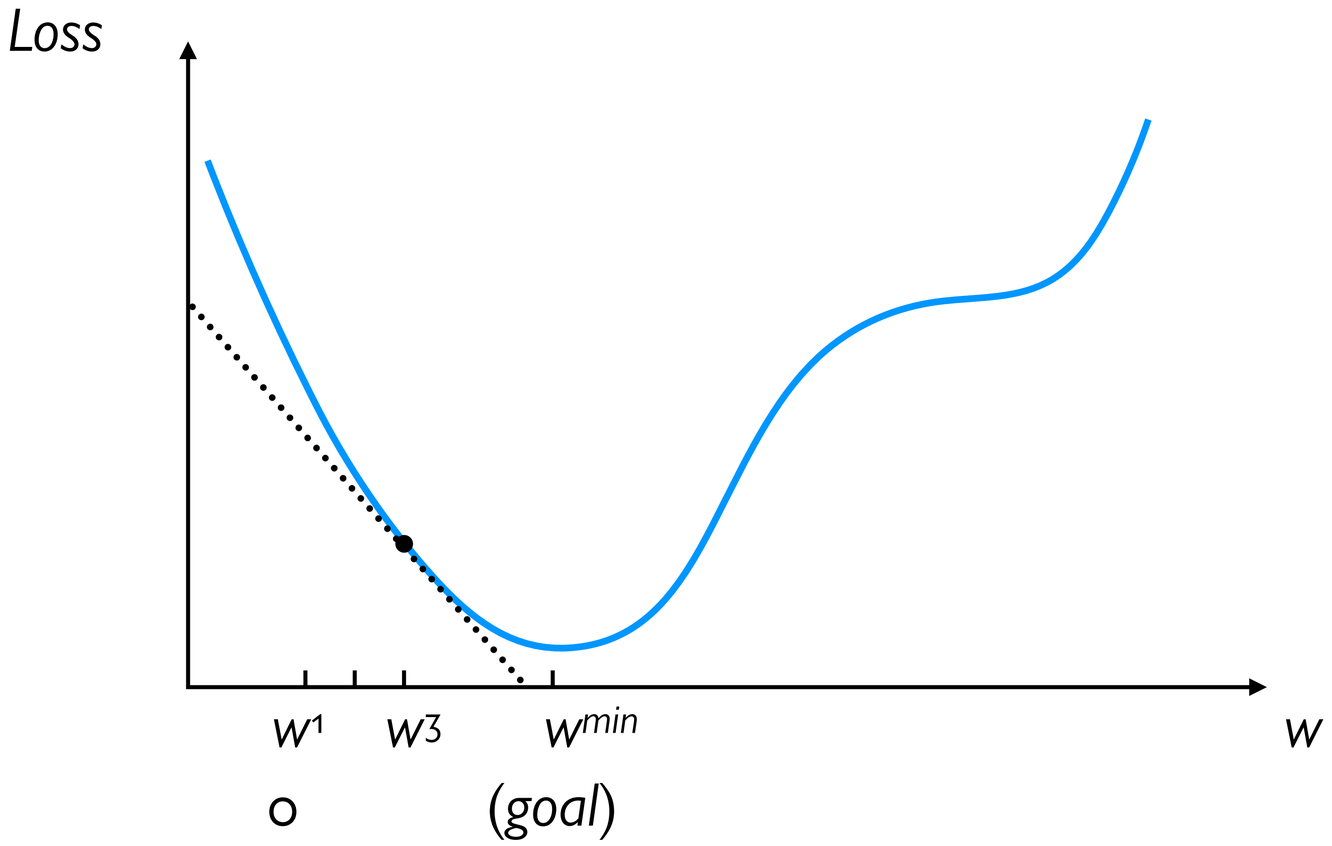

Loss
One step of
gradient descent
w^1
w^{min}
w
0
(goal)

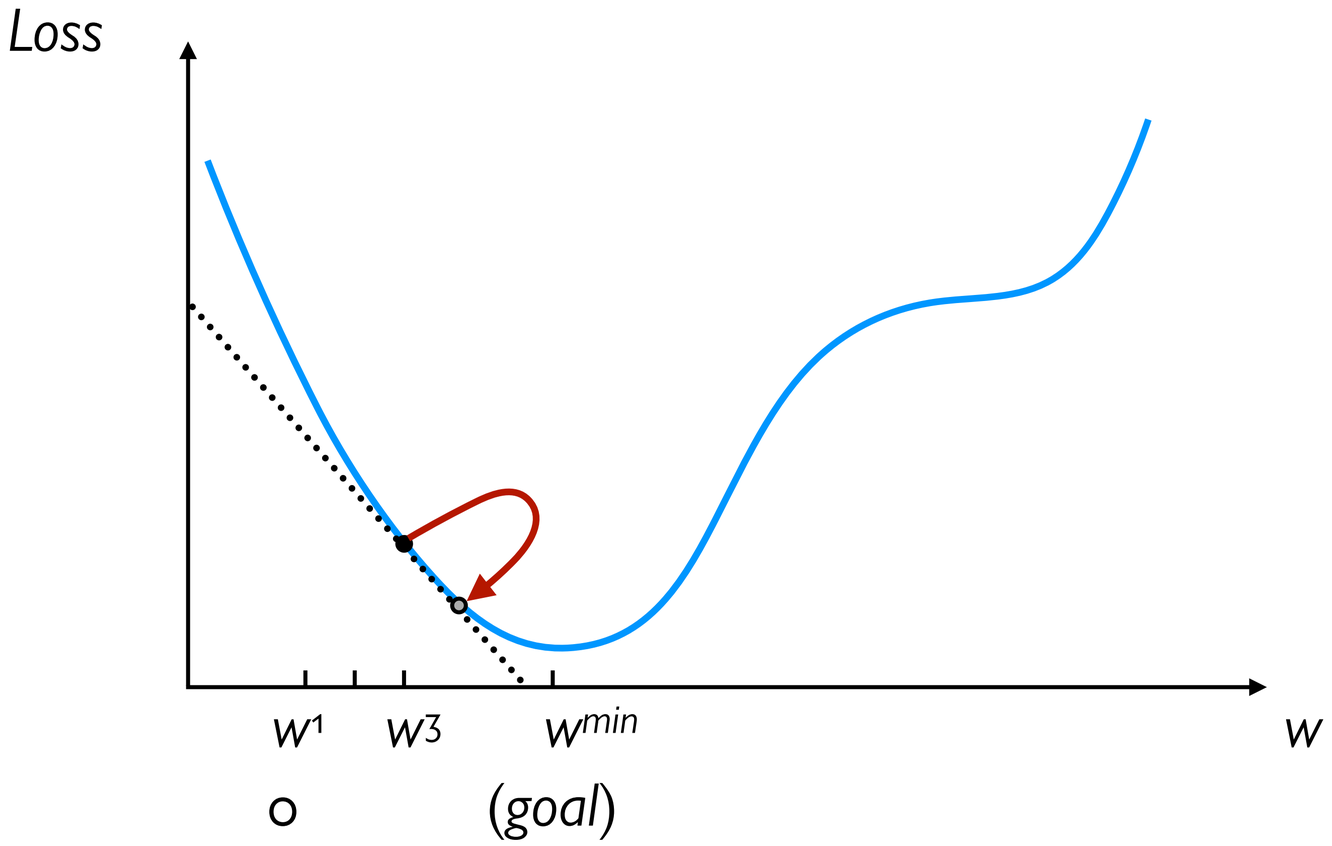

Loss
w
w^1
w^2
w^{min}
0
(goal)

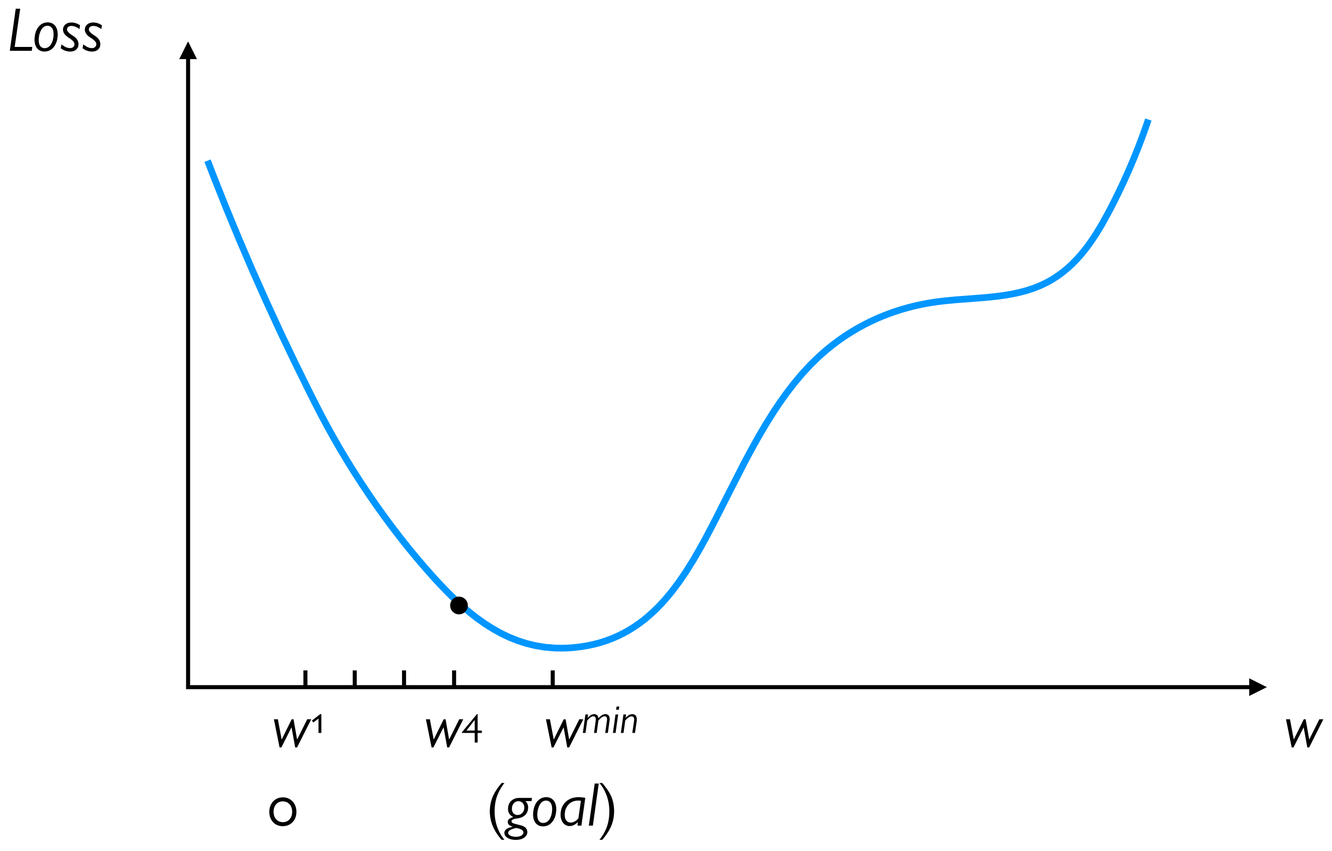

Loss
w
w^1
w^2
w^{min}
0
(goal)

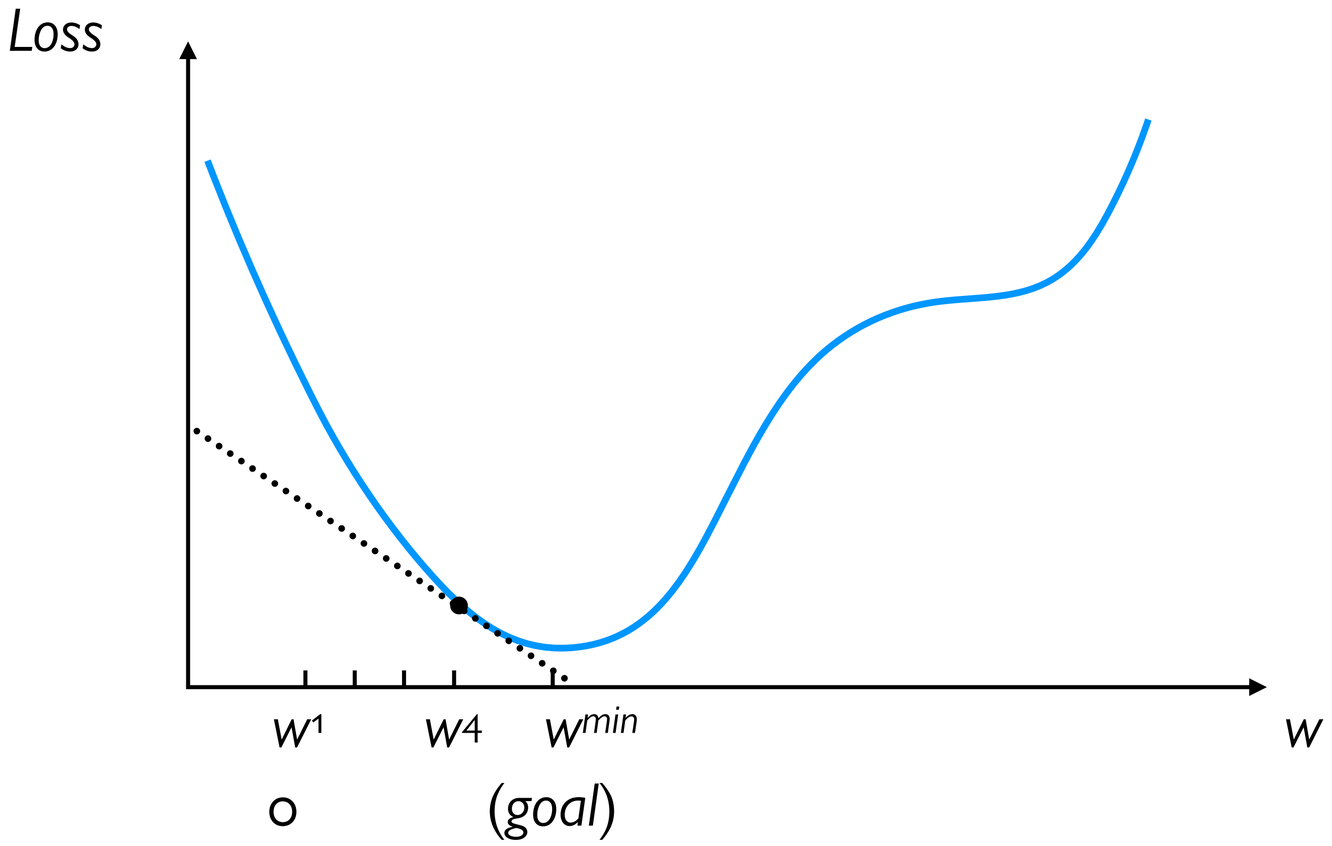

Loss
w
w^1
w^2
w^{min}
0
(goal)

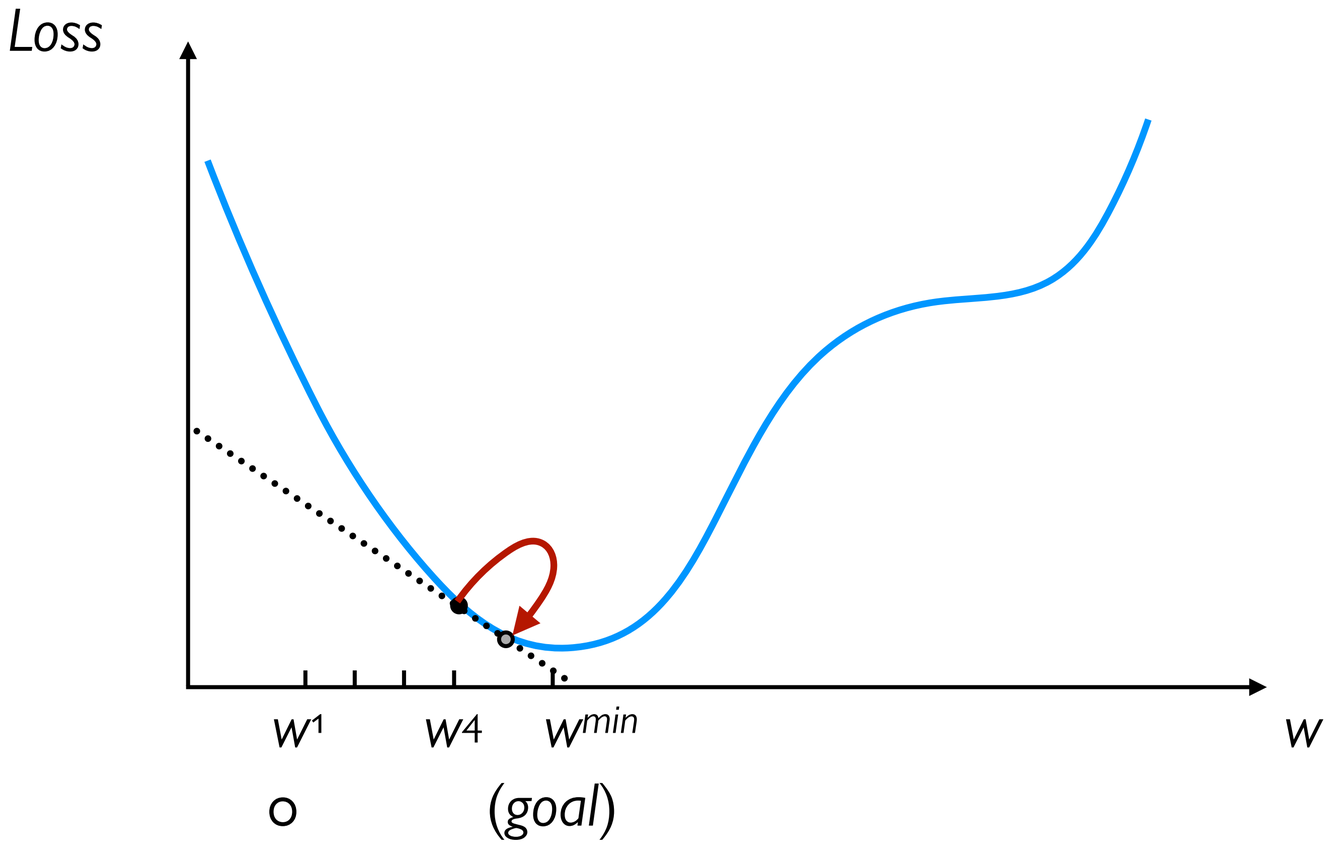

Loss
w
w^1
w^3
w^{min}
0
(goal)

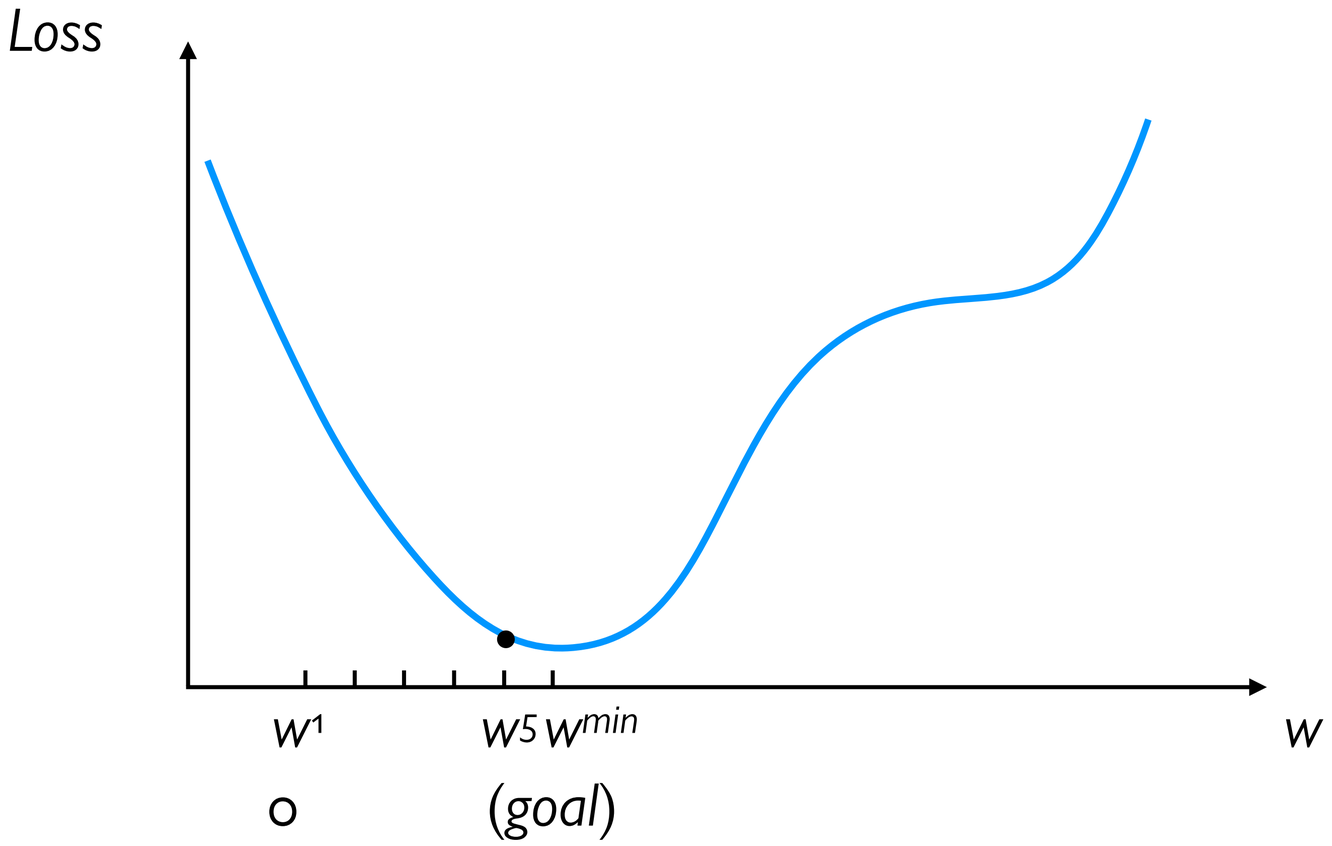

Loss
w
w^1
w^3
w^{min}
0
(goal)

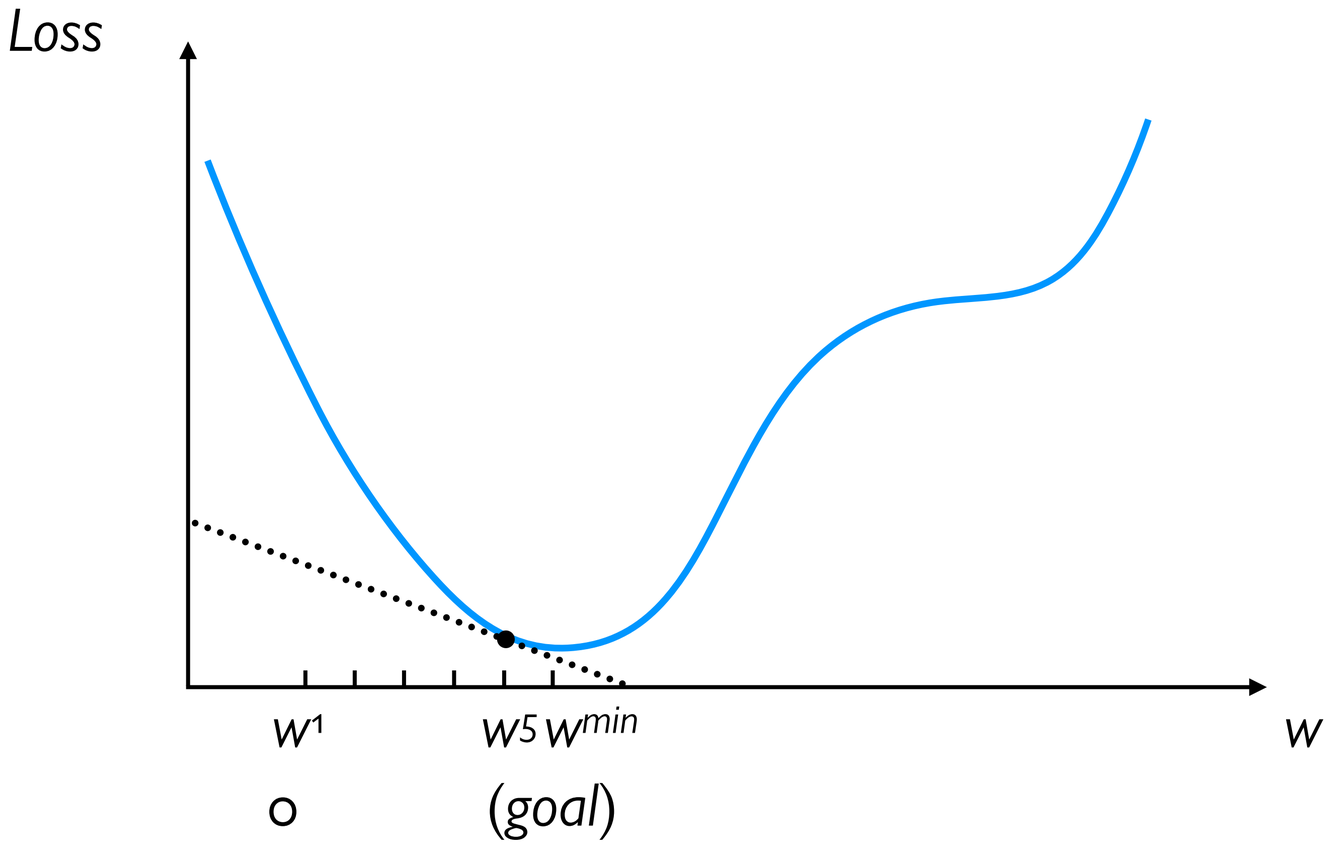

Loss
w
w^1
w^3
w^{min}
0
(goal)

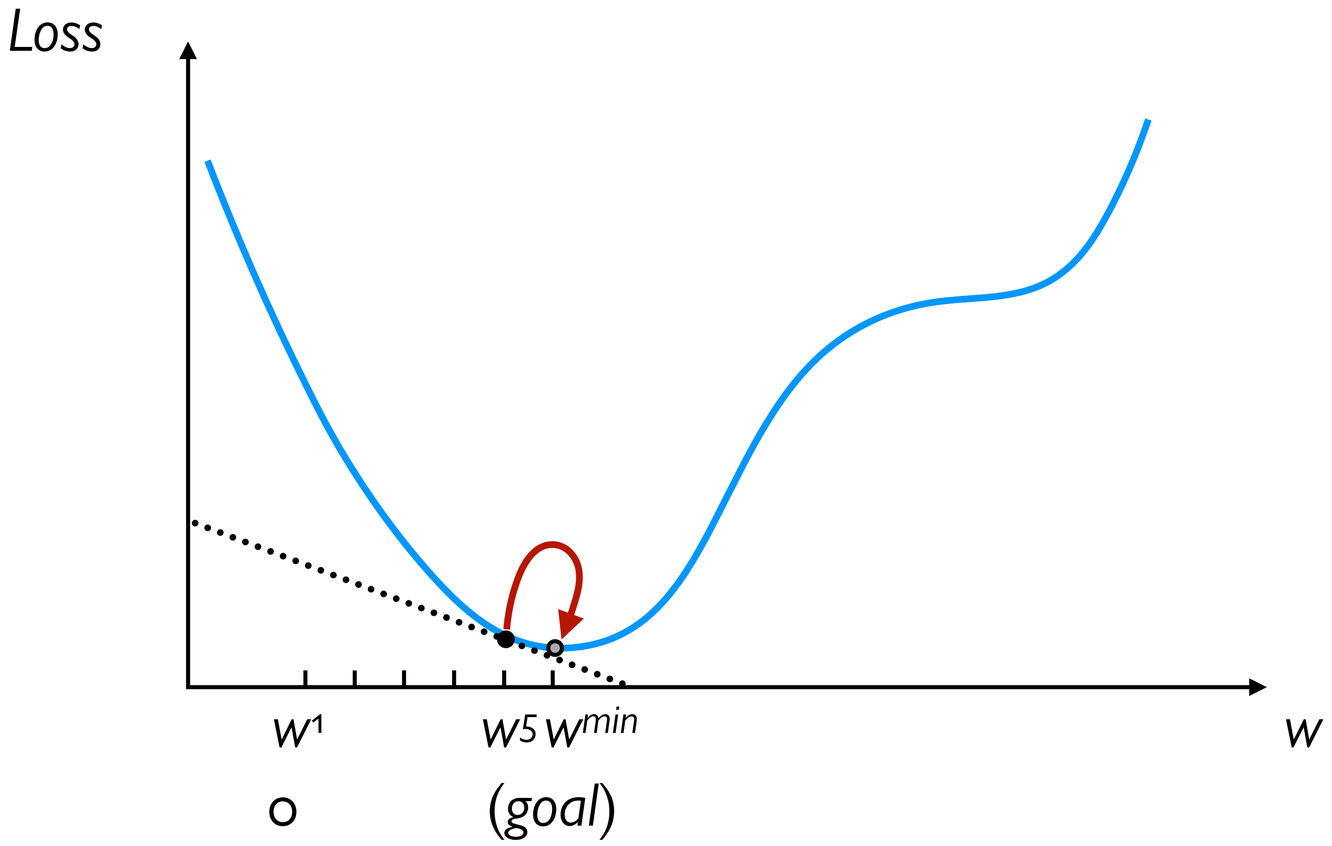

Loss
w
w^1
w^4
w^{min}
0
(goal)

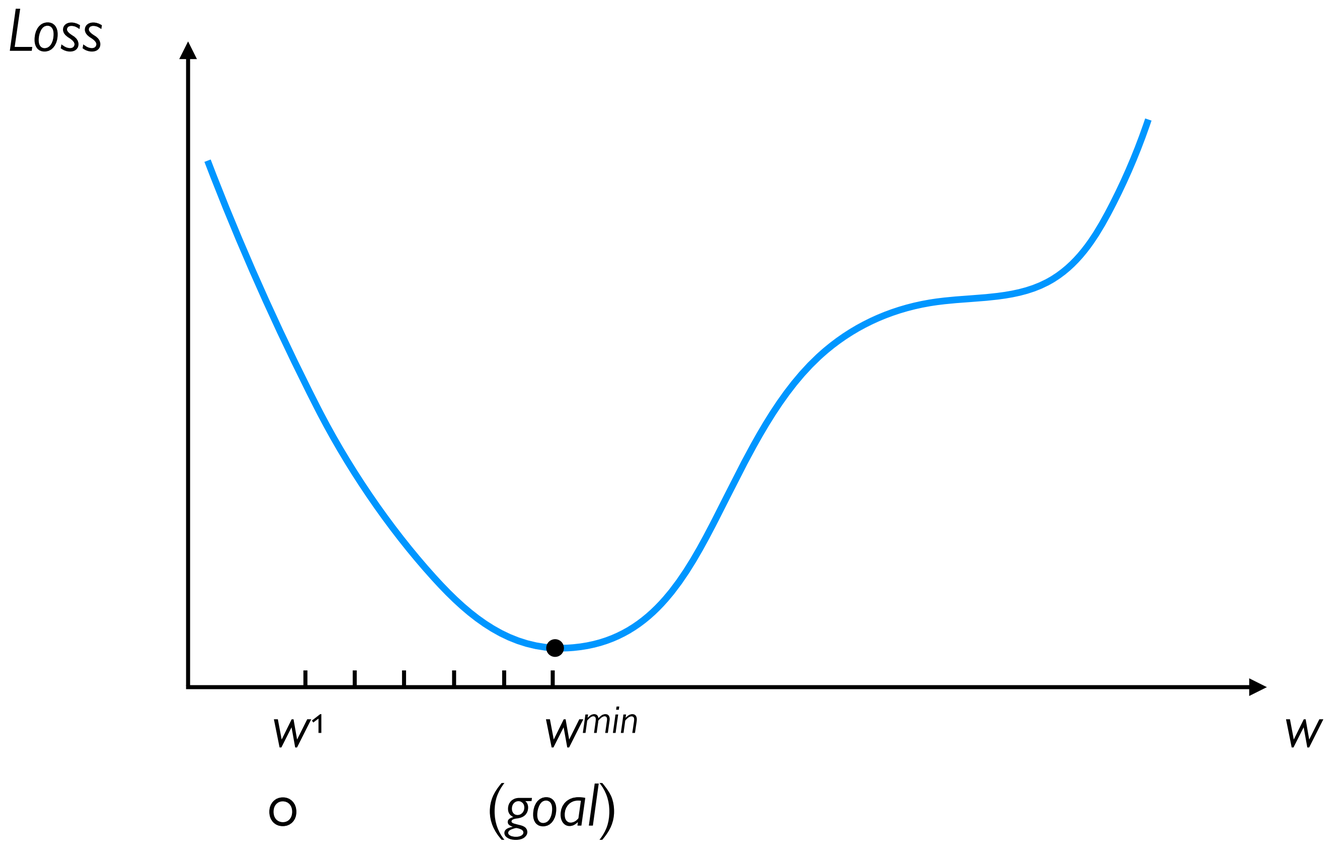

Loss
w
w^1
w^4
w^{min}
0
(goal)

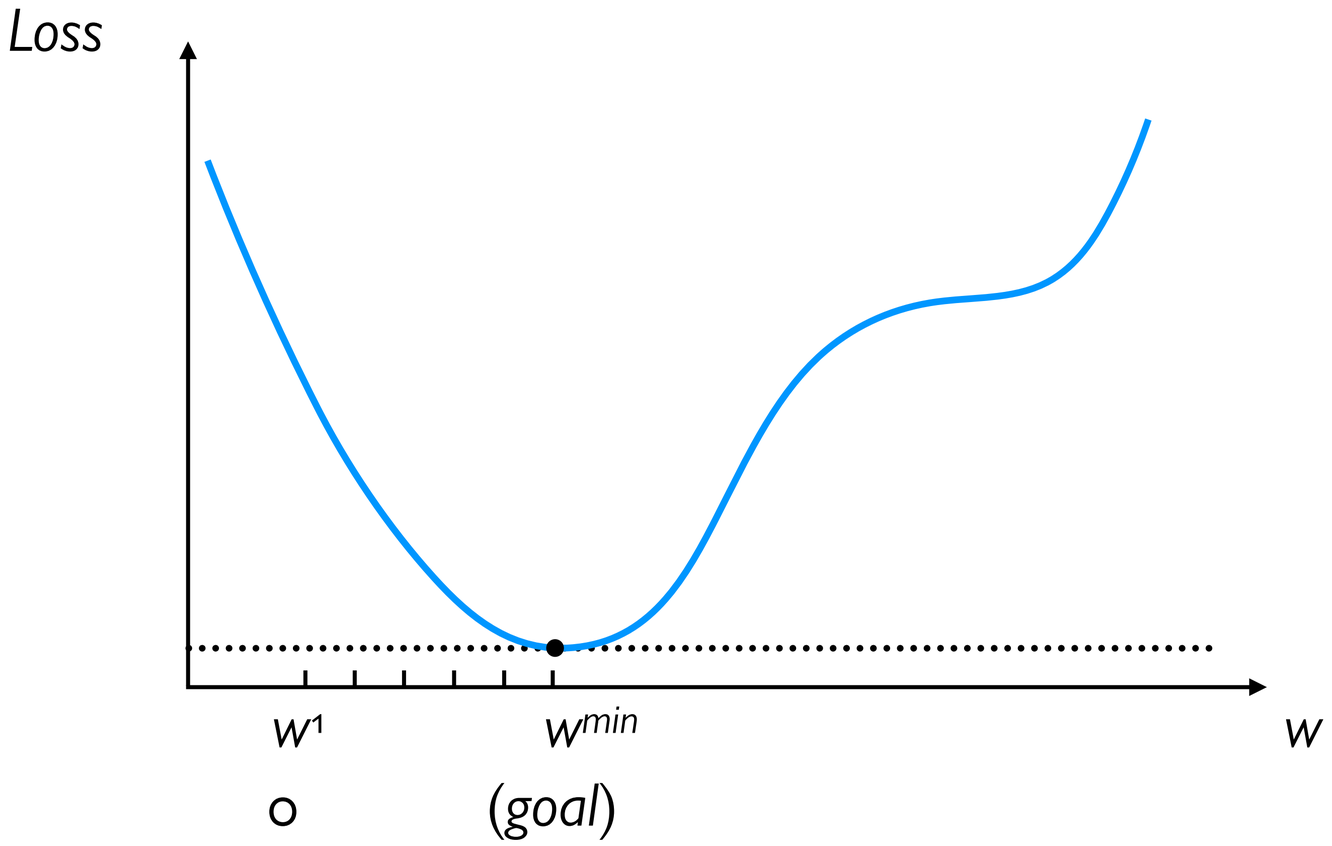

Loss
w
w^1
w^4
w^{min}
0
(goal)


Loss
w
w^1
w^5
w^{min}
0
(goal)


Loss
w
w^1
w^5
w^{min}
0
(goal)


Loss
w
w1
w5
wmin
0
(goal)


Loss
w
w^1
w^{min}
0
(goal)



To go further would be foolish.

For logistic regression, the loss function is *convex*.

A convex function has just one minimum, so gradient descent starting from any point is guaranteed to find the minimum.

The *gradient* of a function of many variables is a vector pointing in the direction of the greatest increase in a function.

The gradient descent algorithm works by finding the gradient of the loss function at the current point and moving in the *opposite* direction.

Gradient descent takes the slope,

$$\frac{\mathrm{d}}{\mathrm{d}w} L(f(x; w), y)$$

and multiplies it by a *learning rate* η.

That's a lowercase Greek letter eta

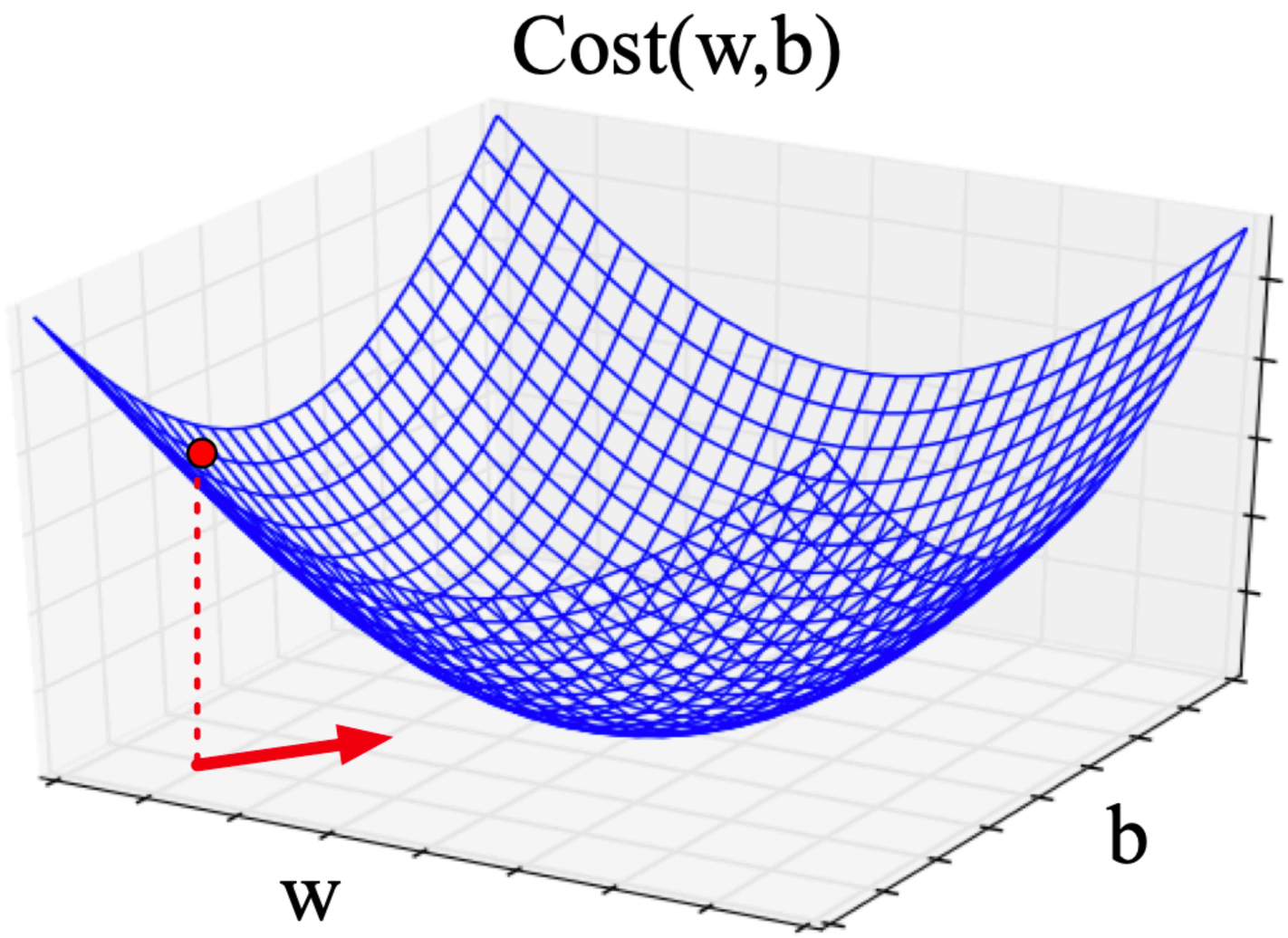
A higher learning rate means that we make bigger adjustments to the weights each time.

So, at time t we calculate the weights for time $t + 1$:

$$w^{t+1} = w^t - \eta \frac{\mathrm{d}}{\mathrm{d}w} L(f(x; w), y)$$

That was for a scalar, but we actually have N parameters making up θ, so we need to know where to move in an N-dimensional space!

The ***gradient*** is just such a vector; it expresses the directional components of the sharpest slope along each of the N dimensions.



Imagine we just add one more parameter – now we have a scalar w and a scalar b.

We'll have more dimensions, making it harder to visualize – but the idea will remain the same: Nudge each of the parameters in the direction that minimizes the loss.

For each dimension w_i, the gradient component i tells us the slope with respect to that variable.

"How much would a small change in w_i influence the total loss function L?"

We express the slope as a partial derivative ∂ of the loss ∂w_i.

The gradient is then defined as the vector of these partials.

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

```
# where: L is the loss function
#        f is a function parameterized by θ
#        x is the set of training inputs x^(1), x^(2),..., x^(m)
#        y is the set of training outputs (labels) y^(1), y^(2),..., y^(m)

θ ← 0
repeat til done
   For each training tuple (x^(i), y^(i)) (in random order)
      1. Optional (for reporting):          # How are we doing on this tuple?
         Compute ŷ^(i) = f(x^(i); θ)        # What is our estimated output ŷ?
         Compute the loss L(ŷ^(i), y^(i))   # How far off is ŷ^(i)) from the true output y^(i)?
      2. g ← ∇_θ L(f(x^(i); θ), y^(i))      # How should we move θ to maximize loss?
      3. θ ← θ − η g                        # Go the other way instead
return θ
```

The learning rate η is a *hyperparameter*.

Too high: The learner will take big steps and overshoot

Too low: The learner will take too long

Hyperparameters are chosen by the algorithm designer instead of being learned from the data like the regular parameters are.

Mini-batch training

Stochastic gradient descent chooses a single random example at a time.

That can result in choppy movements.

It's more common to compute the gradient over batches of training instances.

Batch training: entire dataset

Mini-batch training: *m* examples (e.g., 512 or 1024)

Overfitting and regularization

If a model perfectly matches the training data, that's actually not good.

It will *overfit* the data, modeling noise:

- A random word (maybe a typo) that perfectly predicts *y* because it only occurs in one class will get a very high weight.
- The resulting model will fail to generalize to a test set without this word

This movie drew me in, and it'll do the same to you.

I can't tell you how much I hated this movie. It sucked.

Useful (or, at least, harmless) features:

x_1 = *this*

x_2 = *movie*

x_3 = *hated*

x_4 = *drew me in*

Overfitting

x_5 = *the same to you*

x_6 = *tell you how much*

To avoid overfitting, we use a regularization term, which penalizes large weights that might come from these spurious associations.

See the reading for details!

Evaluation

After choosing the parameters for the classifier – i.e., training it – we test how well it does on a ***test set*** of examples that weren't used for training.

arxiv.org/pdf/2011.08903.pdf

Towards Olfactory Information Extraction from Text: A Case Study on Detecting Smell Experiences in Novels

Ryan Brate and **Paul Groth**
University of Amsterdam
Amsterdam, the Netherlands
r.brate@gmail.com
p.t.groth@uva.nl

Marieke van Erp
KNAW Humanities Cluster
Digital Humanities Lab
Amsterdam, the Netherlands
marieke.van.erp@dh.huc.knaw.nl

Abstract

Environmental factors determine the smells we perceive, but societal factors factors shape the importance, sentiment and biases we give to them. Descriptions of smells in text, or as we call them 'smell experiences', offer a window into these factors, but they must first be identified. To the best of our knowledge, no tool exists to extract references to smell experiences from text. In this paper, we present two variations on a semi-supervised approach to identify smell experiences in English literature. The combined set of patterns from both implementations offer significantly better performance than a keyword-based baseline.

1 Introduction

We rely on our senses: … e another in shaping our

Is a given passage from a book a "smell experience" or not?

You build a “smell” detector

Positive class: Paragraph that involves a smell experience

Negative class: All other paragraphs

The 2×2 confusion matrix

	Gold +	Gold −
Predict +	true positive	false positive
Predict −	false negative	true negative

Evaluation metric: Accuracy

Accuracy is the percent of examples the system labels correctly (both positive and negative).

$$\textit{accuracy} = \frac{\textit{true positives} + \textit{true negatives}}{\textit{true positives} + \textit{false positives} + \textit{true negatives} + \textit{false negatives}}$$

$$= \frac{\textit{true positives} + \textit{true negatives}}{\textit{number of examples}}$$

Evaluation metric: Accuracy

Accuracy sounds great – it considers how the classifier does on *all* inputs!

But 99.99% accuracy might be terrible.

- Imagine we saw 1 million paragraphs and only 100 of them mention smells, we could just label every paragraph as "not about smell".
- But the whole point of the classifier is to help literary scholars find passages about smell to study – so this classifier is useless!

That's why we use precision and recall instead.

Evaluation metric: Precision

Precision is the percent of items the system detected (i.e., labeled +) that are, in fact, positive (according to the human gold labels).

$$precision = \frac{true\ positives}{true\ positives + false\ positives}$$

Evaluation metric: Recall

Recall is the percent of items actually present in the input that were correctly identified by the system.

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

For our classifier that labels nothing as being “about smell”, we get 99.99% accuracy – but 0% recall!

It doesn’t identify any of the 100 paragraphs we wanted.

There's a trade-off between precision and recall.

A highly precise classifier will ignore cases where it's less confident, leading to more false negatives

→ lower recall

A high-recall classifier will flag things it's unsure about, leading to more false positives

→ lower precision

In developing a real application, picking the right trade-off point between precision and recall is an important usability issue.

- Think about a grammar checker: Too many false positives will irritate lots of users.
- But if you're designing a system to detect hate speech online, you might want to err on the side of high recall to avoid abuse slipping through the cracks.

Any balance of precision and recall can be encoded as a single measure called an *F-score*:

$$F_\beta = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

We almost always use balanced F_1, which is the harmonic mean of precision and recall:

$$F_1 = \frac{2PR}{P + R}$$

Why do we use the harmonic mean rather than the mean?

Evaluation with more than two classes

3×3 confusion matrix

System labels \ Gold labels	urgent	normal	spam	
urgent	8	10	1	*precision* $\frac{8}{8+10+1}$
normal	5	60	50	*precision* $\frac{60}{5+60+50}$
spam	3	30	200	*precision* $\frac{200}{3+30+200}$
	recall $\frac{8}{8+5+3}$	*recall* $\frac{60}{10+60+30}$	*recall* $\frac{200}{1+50+200}$	

How can we combine the precision or recall scores from three (or more) classes to get one metric?

Macroaveraging

Compute the performance for each class and then average over classes

Microaveraging

Collect decisions for all classes into one confusion matrix

Compute precision and recall from that table

Macroaveraging and Microaveraging

Class 1: Urgent

	true urgent	true not
system urgent	8	11
system not	8	340

$$\text{precision} = \frac{8}{8+11} = .42$$

Class 2: Normal

	true normal	true not
system normal	60	55
system not	40	212

$$\text{precision} = \frac{60}{60+55} = .52$$

Class 3: Spam

	true spam	true not
system spam	200	33
system not	51	83

$$\text{precision} = \frac{200}{200+33} = .86$$

Pooled

	true yes	true no
system yes	268	99
system no	99	635

$$\text{microaverage precision} = \frac{268}{268+99} = 0.73$$

$$\text{macroaverage precision} = \frac{.42+.52+.86}{3} = 0.60$$

Avoiding harms in classification

Harms in sentiment classifiers

Kiritchenko and Mohammad (2018) found that most sentiment classifiers assigned lower sentiment and more negative emotion to sentences with African American names in them.

This perpetuates negative stereotypes that associate African Americans with negative emotions.

Harms in toxicity classification

Toxicity detection is the task of identifying hate speech, abuse, harassment, and other kinds of toxic language.

But some toxicity classifiers incorrectly flag as being toxic sentences that are non-toxic but simply mention identities like blind people, women, or gay people.

This could lead to censorship of discussion about these groups.

Performance disparities

Text classifiers perform worse on many languages of the world due to lack of data or labels.

Text classifiers perform worse on many varieties of even high-resource languages like English.

What causes these harms?

Can be caused by:

- Problems in the training data; machine learning systems are known to amplify the biases in their training data.
- Problems in the human labels
- Problems in the resources used (like lexicons)
- Problems in model architecture (like what the model is trained to optimize)

Mitigation of these harms is an open research area.

Acknowledgments

This class incorporates material from:

Jurafsky & Martin, *Speech and Language Processing*, 3rd ed. draft

Carolyn Anderson, Wellesley College