



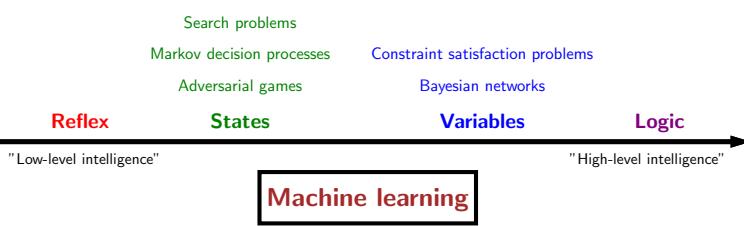
Lecture 2: Machine learning I



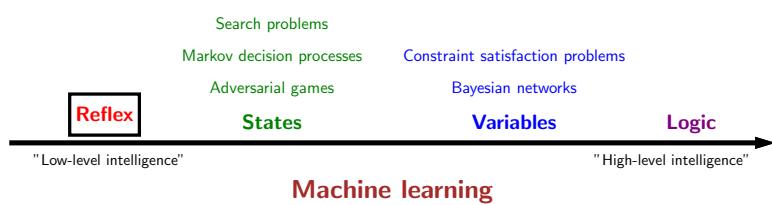
Reminders

- Section: Thursday 3:30-4:20 pm PT, overview of foundations
- Homework is out, due next Tuesday **11 pm** PT

Course plan



Course plan



Roadmap

Linear predictors

Loss minimization

Stochastic gradient descent

- We now embark on our journey into machine learning with the simplest yet most practical tool: **linear predictors**, which cover both classification and regression and are examples of reflex models.
- After getting some geometric intuition for linear predictors, we will turn to learning the weights of a linear predictor by formulating an optimization problem based on the **loss minimization** framework.
- Finally, we will discuss **stochastic gradient descent**, an efficient algorithm for optimizing (that is, minimizing) the loss that's tailored for machine learning which is much faster than gradient descent.

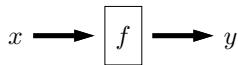
Application: spam classification

Input: x = email message

From: cbfinn@cs.stanford.edu Date: April 4, 2020 Subject: CS221 announcement Hello students, Welcome to CS221! Here's what...	From: a9k62n@hotmail.com Date: April 4, 2020 Subject: URGENT Dear Sir or maDam: my friend left sum of 10m dollars...
---	--

Output: $y \in \{\text{spam, not-spam}\}$

Objective: obtain a **predictor** f



- First, some terminology. A **predictor** is a function f that maps an **input** x to an **output** y . In statistics, y is known as a response, and when x is a real vector, it is known as the covariate.

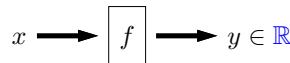
6

Types of prediction tasks

Binary classification (e.g., email \Rightarrow spam/not spam):



Regression (e.g., location, year \Rightarrow housing price):

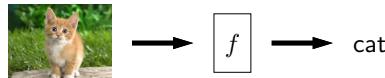


- In the context of classification tasks, f is called a **classifier** and y is called a **label** (sometimes class, category, or tag). The key distinction between binary classification and regression is that the former has **discrete** outputs (e.g., "yes" or "no"), whereas the latter has **continuous** outputs.
- Note that the dichotomy of prediction tasks are not meant to be formal definitions, but rather to provide intuition.
- For instance, binary classification could technically be seen as a regression problem if the labels are -1 and $+1$. And structured prediction generally refers to tasks where the possible set of outputs y is huge (generally, exponential in the size of the input), but where each individual y has some structure. For example, in machine translation, the output is a sequence of words.

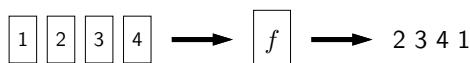
8

Types of prediction tasks

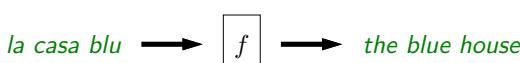
Multiclass classification: y is a category



Ranking: y is a permutation



Structured prediction: y is an object which is built from parts



Data

Example: specifies that y is the ground-truth output for x

$$(x, y)$$

Training data: list of examples

$$\mathcal{D}_{\text{train}} = [$$

$$("...10m dollars...", +1),$$

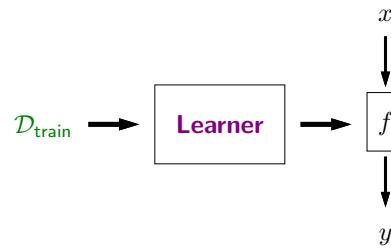
$$("...CS221...", -1),$$

$$]$$

partial specification of behavior

- The starting point of machine learning is the data.
- For now, we will focus on **supervised learning**, in which our data provides both inputs and outputs, in contrast to unsupervised learning, which only provides inputs.
- A (supervised) **example** (also called a data point or instance) is simply an input-output pair (x, y) , which specifies that y is the ground-truth output for x .
- The **training data** $\mathcal{D}_{\text{train}}$ is a multiset of examples (repeats are allowed, but this is not important), which forms a partial specification of the desired behavior of a predictor.

Framework



CS221 / Spring 2020 / Finn & Anari

13

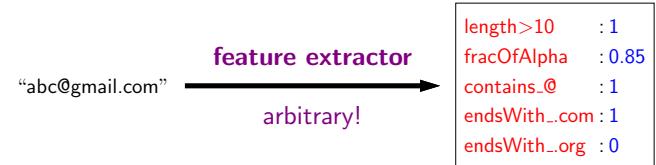
- Learning** is about taking the training data $\mathcal{D}_{\text{train}}$ and producing a predictor f , which is a function that takes inputs x and tries to map them to outputs $y = f(x)$. One thing to keep in mind is that we want the predictor to approximately work even for examples that we have not seen in $\mathcal{D}_{\text{train}}$. This problem of generalization, which we will discuss two lectures from now, forces us to design f in a principled, mathematical way.
- We will first focus on examining what f is, independent of how the learning works. Then we will come back to learning f based on data.

Feature extraction

Example task: predict y , whether a string x is an email address

Question: what properties of x **might be** relevant for predicting y ?

Feature extractor: Given input x , output a set of (**feature name**, **feature value**) pairs.



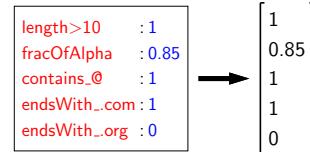
CS221 / Spring 2020 / Finn & Anari [features]

15

- We will consider predictors f based on **feature extractors**. Feature extraction is a bit of an art that requires intuition about both the task and also what machine learning algorithms are capable of.
- The general principle is that features should represent properties of x which **might be** relevant for predicting y . It is okay to add features which turn out to be irrelevant, since the learning algorithm can sort it out (though it might require more data to do so).

Feature vector notation

Mathematically, feature vector doesn't need feature names:



Definition: feature vector

For an input x , its feature vector is:

$$\phi(x) = [\phi_1(x), \dots, \phi_d(x)].$$

Think of $\phi(x) \in \mathbb{R}^d$ as a point in a high-dimensional space.

CS221 / Spring 2020 / Finn & Anari

17

- Each input x represented by a **feature vector** $\phi(x)$, which is computed by the feature extractor ϕ . When designing features, it is useful to think of the feature vector as being a map from strings (feature names) to doubles (feature values). But formally, the feature vector $\phi(x) \in \mathbb{R}^d$ is a real vector $\phi(x) = [\phi_1(x), \dots, \phi_d(x)]$, where each component $\phi_j(x)$, for $j = 1, \dots, d$, represents a feature.
- This vector-based representation allows us to think about feature vectors as a point in a (high-dimensional) vector space, which will later be useful for getting some geometric intuition.

Weight vector

Weight vector: for each feature j , have real number w_j representing contribution of feature to prediction

length>10	:-1.2
fracOfAlpha	:0.6
contains_@	:3
endsWith..com	:2.2
endsWith..org	:1.4
...	

- So far, we have defined a feature extractor ϕ that maps each input x to the feature vector $\phi(x)$. A **weight vector** $w = [w_1, \dots, w_d]$ (also called a parameter vector or weights) specifies the contributions of each feature vector to the prediction.
- In the context of binary classification with binary features ($\phi_j(x) \in \{0, 1\}$), the weights $w_j \in \mathbb{R}$ have an intuitive interpretation. If w_j is positive, then the presence of feature j ($\phi_j(x) = 1$) favors a positive classification. Conversely, if w_j is negative, then the presence of feature j favors a negative classification.
- Note that while the feature vector depends on the input x , the weight vector does not. This is because we want a single predictor (specified by the weight vector) that works on any input.

CS221 / Spring 2020 / Finn & Anari

19

Linear predictors

Weight vector $w \in \mathbb{R}^d$ Feature vector $\phi(x) \in \mathbb{R}^d$

length>10	:-1.2
fracOfAlpha	:0.6
contains_@	:3
endsWith..com	:2.2
endsWith..org	:1.4

length>10	:1
fracOfAlpha	:0.85
contains_@	:1
endsWith..com	:1
endsWith..org	:0

Score: weighted combination of features

$$w \cdot \phi(x) = \sum_{j=1}^d w_j \phi(x)_j$$

Example: $-1.2(1) + 0.6(0.85) + 3(1) + 2.2(1) + 1.4(0) = 4.51$

CS221 / Spring 2020 / Finn & Anari

21

Linear predictors

Weight vector $w \in \mathbb{R}^d$

Feature vector $\phi(x) \in \mathbb{R}^d$

For binary classification:

	Definition: (binary) linear classifier
$f_w(x) = \text{sign}(w \cdot \phi(x)) = \begin{cases} +1 & \text{if } w \cdot \phi(x) > 0 \\ -1 & \text{if } w \cdot \phi(x) < 0 \\ ? & \text{if } w \cdot \phi(x) = 0 \end{cases}$	

CS221 / Spring 2020 / Finn & Anari

23

- We now have gathered enough intuition that we can formally define the predictor f . For each weight vector w , we write f_w to denote the predictor that depends on w and takes the sign of the score.
- For the next few slides, we will focus on the case of binary classification. Recall that in this setting, we call the predictor a (binary) classifier.
- The case of $f_w(x) = ?$ is a boundary case that isn't so important. We can just predict +1 arbitrarily as a matter of convention.

Geometric intuition

Example:

$$w = [2, -1]$$

$$\phi(x) \in \{[2, 0], [0, 2], [2, 4]\}$$

[whiteboard]

In general: binary classifier f_w defines a hyperplane **decision boundary** with normal vector w .

\mathbb{R}^2 : hyperplane is a line

\mathbb{R}^3 : hyperplane is a plane

- So far, we have talked about linear predictors as weighted combinations of features. We can get a bit more insight by studying the **geometry** of the problem.
- Let's visualize the predictor f_w by looking at which points it classifies positive. Specifically, we can draw a ray from the origin to w (in two dimensions).
- Points which form an acute angle with w are classified as positive (dot product is positive), and points that form an obtuse angle with w are classified as negative. Points which are orthogonal $\{z \in \mathbb{R}^d : w \cdot z = 0\}$ constitute the **decision boundary**.
- By changing w , we change the predictor f_w and thus the decision boundary as well.

CS221 / Spring 2020 / Finn & Anari

25



Roadmap

Linear predictors

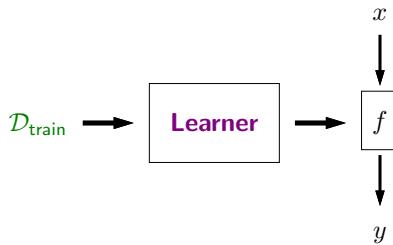
Loss minimization

Stochastic gradient descent

CS221 / Spring 2020 / Finn & Anari

27

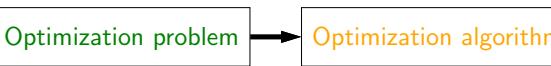
Framework



- So far we have talked about linear predictors f_w which are based on a feature extractor ϕ and a weight vector w . Now we turn to the problem of estimating (also known as fitting or learning) w from training data.

- The **loss minimization** framework is to cast learning as an optimization problem. Note the theme of separating your problem into a model (optimization problem) and an algorithm (optimization algorithm).

Learner



Loss functions



Definition: loss function

A loss function $\text{Loss}(x, y, \mathbf{w})$ quantifies how unhappy you would be if you used \mathbf{w} to make a prediction on x when the correct output is y . It is the object we want to minimize.

Score and margin

Correct label: y

Predicted label: $y' = f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$

Example: $\mathbf{w} = [2, -1], \phi(x) = [2, 0], y = -1$



Definition: score

The score on an example (x, y) is $\mathbf{w} \cdot \phi(x)$, how **confident** we are in predicting +1.



Definition: margin

The margin on an example (x, y) is $(\mathbf{w} \cdot \phi(x))y$, how **correct** we are.

- Before we talk about what loss functions look like and how to learn \mathbf{w} , we introduce another important concept, the notion of a **margin**. Suppose the correct label is $y \in \{-1, +1\}$. The margin of an input x is $\mathbf{w} \cdot \phi(x)y$, which measures how correct the prediction that \mathbf{w} makes is. The larger the margin the better, and non-positive margins correspond to classification errors.
- Note that if we look at the actual prediction $f_{\mathbf{w}}(x)$, we can only ascertain whether the prediction was right or not. By looking at the score and the margin, we can get a more nuanced view into the behavior of the classifier.
- Geometrically, if $\|\mathbf{w}\| = 1$, then the margin of an input x is exactly the distance from its feature vector $\phi(x)$ to the **decision boundary**.



cs221.stanford.edu/q

Question

When does a binary classifier err on an example?

- margin less than 0
- margin greater than 0
- score less than 0
- score greater than 0

Binary classification

Example: $\mathbf{w} = [2, -1], \phi(x) = [2, 0], y = -1$

Recall the **binary classifier**:

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$$

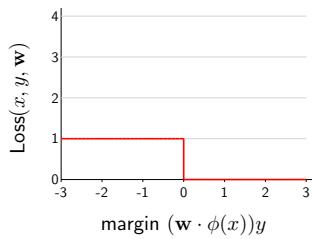


Definition: zero-one loss

$$\begin{aligned} \text{Loss}_{0-1}(x, y, \mathbf{w}) &= \mathbf{1}[f_{\mathbf{w}}(x) \neq y] \\ &= \mathbf{1}[\underbrace{(\mathbf{w} \cdot \phi(x))y}_{\text{margin}} \leq 0] \end{aligned}$$

- Now let us define our first loss function, the **zero-one loss**. This corresponds exactly to our familiar notion of whether our predictor made a mistake or not. We can also write the loss in terms of the margin.

Binary classification

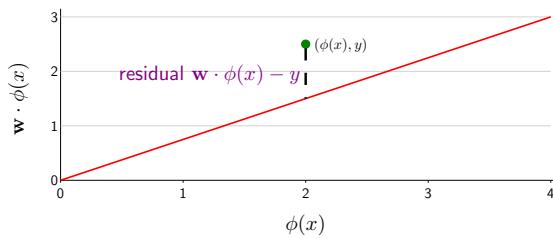


$$\text{Loss}_{0-1}(x, y, w) = \mathbf{1}[(w \cdot \phi(x))y \leq 0]$$

- We can plot the loss as a function of the margin. From the graph, it is clear that the loss is 1 when the margin is negative and 0 when it is positive.

Linear regression

$$f_w(x) = w \cdot \phi(x)$$

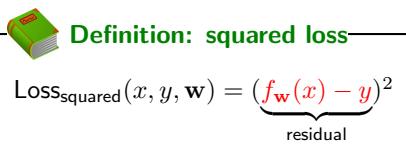


Definition: residual

The **residual** is $(w \cdot \phi(x)) - y$, the amount by which prediction $f_w(x) = w \cdot \phi(x)$ overshoots the target y .

Linear regression

$$f_w(x) = w \cdot \phi(x)$$

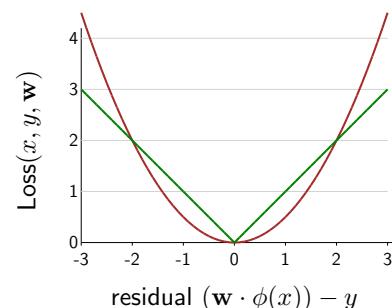


Example:

$$w = [2, -1], \phi(x) = [2, 0], y = -1$$

$$\text{Loss}_{\text{squared}}(x, y, w) = 25$$

Regression loss functions



$$\text{Loss}_{\text{squared}}(x, y, w) = (w \cdot \phi(x) - y)^2$$

$$\text{Loss}_{\text{absdev}}(x, y, w) = |w \cdot \phi(x) - y|$$

- A popular and convenient loss function to use in linear regression is the **squared loss**, which penalizes the residual of the prediction quadratically. If the predictor is off by a residual of 10, then the loss will be 100.
- An alternative to the squared loss is the **absolute deviation loss**, which simply takes the absolute value of the residual.

Loss minimization framework

So far: one example, $\text{Loss}(x, y, \mathbf{w})$ is easy to minimize.



$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

$$\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainLoss}(\mathbf{w})$$

Key: need to set \mathbf{w} to make global tradeoffs — not every example can be happy.

- Note that on one example, both the squared and absolute deviation loss functions have the same minimum, so we cannot really appreciate the differences here. However, we are learning \mathbf{w} based on a whole training set $\mathcal{D}_{\text{train}}$, not just one example. We typically minimize the **training loss** (also known as the training error or empirical risk), which is the average loss over all the training examples.
- Importantly, such an optimization problem requires making tradeoffs across all the examples (in general, we won't be able to set \mathbf{w} to a single value that makes every example have low loss).

CS221 / Spring 2020 / Finn & Anari

43

Which regression loss to use? (skip)

Example: $\mathcal{D}_{\text{train}} = \{(1, 0), (1, 2), (1, 1000)\}$, $\phi(x) = x$

For least squares (L_2) regression:

$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

- \mathbf{w} that minimizes training loss is **mean** y
- **Mean:** tries to accommodate every example, popular

For least absolute deviation (L_1) regression:

$$\text{Loss}_{\text{absdev}}(x, y, \mathbf{w}) = |\mathbf{w} \cdot \phi(x) - y|$$

- \mathbf{w} that minimizes training loss is **median** y
- **Median:** more robust to outliers

CS221 / Spring 2020 / Finn & Anari

45

- Now the question of which loss we should use becomes more interesting.
- For example, consider the case where all the inputs are $\phi(x) = 1$. Essentially the problem becomes one of predicting a single value y^* which is the least offensive towards all the examples.
- If our loss function is the squared loss, then the optimal value is the mean $y^* = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} y$. If our loss function is the absolute deviation loss, then the optimal value is the median.
- The median is more robust to outliers: you can move the furthest point arbitrarily farther out without affecting the median. This makes sense given that the squared loss penalizes large residuals a lot more.
- In summary, this is an example of where the choice of the loss function has a qualitative impact on the weights learned, and we can study these differences in terms of the objective function without thinking about optimization algorithms.



Roadmap

Linear predictors

Loss minimization

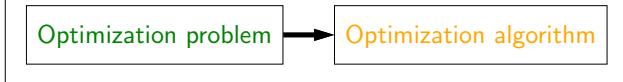
Stochastic gradient descent

CS221 / Spring 2020 / Finn & Anari

47

Learning as optimization

Learner



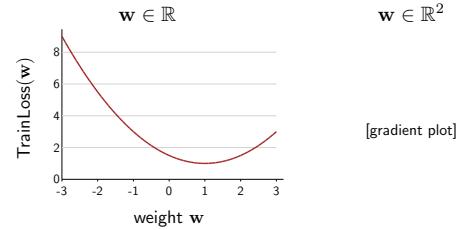
48

CS221 / Spring 2020 / Finn & Anari

- Having defined a bunch of different objective functions that correspond to training loss, we would now like to optimize them — that is, obtain an algorithm that outputs the \mathbf{w} where the objective function achieves the minimum value.

Optimization problem

Objective: $\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainLoss}(\mathbf{w})$



[gradient plot]

49

CS221 / Spring 2020 / Finn & Anari

How to optimize?



Definition: gradient

The gradient $\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$ is the direction that increases the loss the most.



Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \dots, 0]$

For $t = 1, \dots, T$:

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

CS221 / Spring 2020 / Finn & Anari

51

- A general approach is to use **iterative optimization**, which essentially starts at some starting point \mathbf{w} (say, all zeros), and tries to tweak \mathbf{w} so that the objective function value decreases.
- To do this, we will rely on the gradient of the function, which tells us which direction to move in to decrease the objective the most. The gradient is a valuable piece of information, especially since we will often be optimizing in high dimensions (d on the order of thousands).
- This iterative optimization procedure is called **gradient descent**. Gradient descent has two **hyperparameters**, the **step size** η (which specifies how aggressively we want to pursue a direction) and the number of iterations T . Let's not worry about how to set them, but you can think of $T = 100$ and $\eta = 0.1$ for now.

Least squares regression

Objective function:

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (\mathbf{w} \cdot \phi(x) - y)^2$$

Gradient (use chain rule):

$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2 \underbrace{(\mathbf{w} \cdot \phi(x) - y)}_{\text{prediction} - \text{target}} \phi(x)$$

[semi-live solution]

CS221 / Spring 2020 / Finn & Anari

53

- All that's left to do before we can use gradient descent is to compute the gradient of our objective function TrainLoss. The calculus can usually be done by hand; combinations of the product and chain rule suffice in most cases for the functions we care about.
- Note that the gradient often has a nice interpretation. For squared loss, it is the residual (prediction - target) times the feature vector $\phi(x)$.
- Note that for linear predictors, the gradient is always something times $\phi(x)$ because w only affects the loss through $w \cdot \phi(x)$.



Gradient descent is slow

$$\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, w)$$

Gradient descent:

$$w \leftarrow w - \eta \nabla_w \text{TrainLoss}(w)$$

Problem: each iteration requires going over all training examples — expensive when have lots of data!

- We can now apply gradient descent on any of our objective functions that we defined before and have a working algorithm. But it is not necessarily the best algorithm.
- One problem (but not the only problem) with gradient descent is that it is slow. Those of you familiar with optimization will recognize that methods like Newton's method can give faster convergence, but that's not the type of slowness I'm talking about here.
- Rather, it is the slowness that arises in large-scale machine learning applications. Recall that the training loss is a sum over the training data. If we have one million training examples (which is, by today's standards, only a modest number), then each gradient computation requires going through those one million examples, and this must happen before we can make any progress. Can we make progress before seeing all the data?

CS221 / Spring 2020 / Finn & Anari

55



Stochastic gradient descent

$$\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, w)$$

Gradient descent (GD):

$$w \leftarrow w - \eta \nabla_w \text{TrainLoss}(w)$$

Stochastic gradient descent (SGD):

For each $(x, y) \in \mathcal{D}_{\text{train}}$:

$$w \leftarrow w - \eta \nabla_w \text{Loss}(x, y, w)$$



Key idea: stochastic updates

It's not about **quality**, it's about **quantity**.

- The answer is **stochastic gradient descent** (SGD). Rather than looping through all the training examples to compute a single gradient and making one step, SGD loops through the examples (x, y) and updates the weights w based on **each** example. Each update is not as good because we're only looking at one example rather than all the examples, but we can make many more updates this way.
- In practice, we often find that just performing one pass over the training examples with SGD, touching each example once, often performs comparably to taking ten passes over the data with GD.
- There are other variants of SGD. You can randomize the order in which you loop over the training data in each iteration, which is useful. Think about what would happen if you had all the positive examples first and the negative examples after that.

CS221 / Spring 2020 / Finn & Anari [stochastic gradient descent]

57



Stochastic gradient descent

$$\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, w)$$

Gradient descent (GD):

$$w \leftarrow w - \eta \nabla_w \text{TrainLoss}(w)$$

Stochastic gradient descent (SGD):

For each $(x, y) \in \mathcal{D}_{\text{train}}$:

$$w \leftarrow w - \eta \nabla_w \text{Loss}(x, y, w)$$

[semi-live solution]

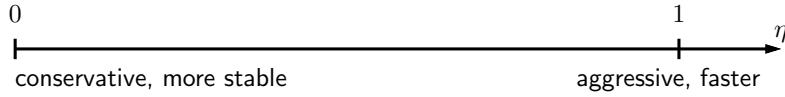
CS221 / Spring 2020 / Finn & Anari [stochastic gradient descent]

59

Step size

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

Question: what should η be?



Strategies:

- Constant: $\eta = 0.1$
- Decreasing: $\eta = 1/\sqrt{\# \text{ updates made so far}}$

Summary so far

Linear predictors:

$$f_{\mathbf{w}}(x) \text{ based on score } \mathbf{w} \cdot \phi(x)$$

Loss minimization: learning as optimization

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

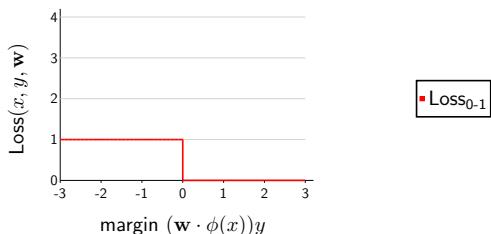
Stochastic gradient descent: optimization algorithm

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

Done for linear regression; what about classification?

Zero-one loss

$$\text{Loss}_{0-1}(x, y, \mathbf{w}) = \mathbf{1}[(\mathbf{w} \cdot \phi(x))y \leq 0]$$



Problems:

- Gradient of Loss_{0-1} is 0 everywhere, SGD not applicable
- Loss_{0-1} is insensitive to how badly the model messed up

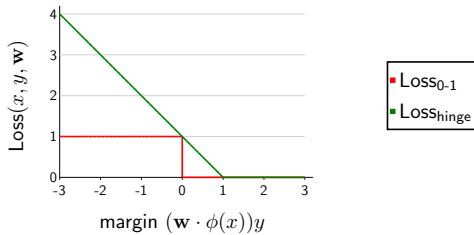
- One remaining issue is choosing the step size, which in practice (and as we have seen) is actually quite important. Generally, larger step sizes are like driving fast. You can get faster convergence, but you might also get very unstable results and crash and burn. On the other hand, with smaller step sizes you get more stability, but you might get to your destination more slowly.
- A suggested form for the step size is to set the initial step size to 1 and let the step size decrease as the inverse of the square root of the number of updates we've taken so far. There are some nice theoretical results showing that SGD is guaranteed to converge in this case (provided all your gradients have bounded length).

- In summary we have seen (i) the functions we're considering (linear predictors), (ii) the criterion for choosing one (loss minimization), and (iii) an algorithm that goes after that criterion (SGD).
- We already worked out a linear regression example. What are good loss functions for binary classification?

- Recall that we have the zero-one loss for classification. But the main problem with zero-one loss is that it's hard to optimize (in fact, it's provably NP hard in the worst case). And in particular, we cannot apply gradient-based optimization to it, because the gradient is zero (almost) everywhere.

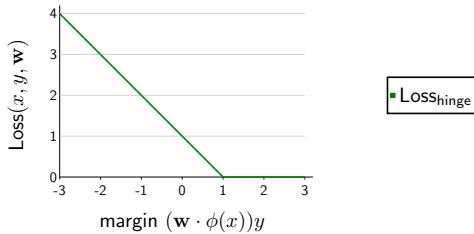
Hinge loss (SVMs)

$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$



- **Intuition:** hinge loss upper bounds 0-1 loss, has non-trivial gradient
- Try to increase margin if it is less than 1

A gradient exercise



Problem: Gradient of hinge loss

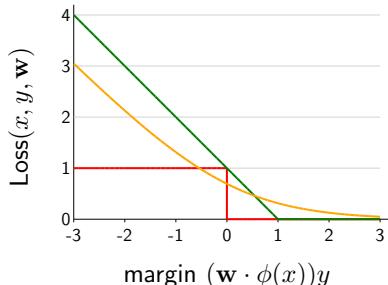
Compute the gradient of

$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

[whiteboard]

Logistic regression

$$\text{Loss}_{\text{logistic}}(x, y, \mathbf{w}) = \log(1 + e^{-(\mathbf{w} \cdot \phi(x))y})$$



- **Intuition:** Try to increase margin even when it already exceeds 1

- To fix this problem, we can use the **hinge loss**, which is an upper bound on the zero-one loss. Minimizing upper bounds are a general idea; the hope is that pushing down the upper bound leads to pushing down the actual function.
- Advanced: The hinge loss corresponds to the **Support Vector Machine (SVM)** objective function with one important difference. The SVM objective function also includes a **regularization penalty** $\|\mathbf{w}\|^2$, which prevents the weights from getting too large. We will get to regularization later in the course, so you needn't worry about this for now. But if you're curious, read on.
- Why should we penalize $\|\mathbf{w}\|^2$? One answer is Occam's razor, which says to find the simplest hypothesis that explains the data. Here, simplicity is measured in the length of \mathbf{w} . This can be made formal using statistical learning theory (take CS229T if you want to learn more).
- Perhaps a less abstract and more geometric reason is the following. Recall that we defined the (algebraic) margin to be $\mathbf{w} \cdot \phi(x)y$. The actual (signed) distance from a point to the decision boundary is actually $\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \phi(x)y$ — this is called the geometric margin. So the loss being zero (that is, $\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = 0$) is equivalent to the algebraic margin being at least 1 (that is, $\mathbf{w} \cdot \phi(x)y \geq 1$), which is equivalent to the geometric margin being larger than $\frac{1}{\|\mathbf{w}\|}$ (that is, $\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \phi(x)y \geq \frac{1}{\|\mathbf{w}\|}$). Therefore, reducing $\|\mathbf{w}\|$ increases the geometric margin. For this reason, SVMs are also referred to as max-margin classifiers.

- You should try to "see" the solution before you write things down formally. Pictorially, it should be evident: when the margin is less than 1, then the gradient is the gradient of $1 - (\mathbf{w} \cdot \phi(x))y$, which is equal to $-\phi(x)y$. If the margin is larger than 1, then the gradient is the gradient of 0, which is 0. Combining the two cases: $\nabla_{\mathbf{w}} \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \begin{cases} -\phi(x)y & \text{if } \mathbf{w} \cdot \phi(x)y < 1 \\ 0 & \text{if } \mathbf{w} \cdot \phi(x)y > 1. \end{cases}$
- What about when the margin is exactly 1? Technically, the gradient doesn't exist because the hinge loss is not differentiable there. Fear not! Practically speaking, at the end of the day, we can take either $-\phi(x)y$ or 0 (or anything in between).
- Technical note (can be skipped): given $f(\mathbf{w})$, the gradient $\nabla f(\mathbf{w})$ is only defined at points \mathbf{w} where f is differentiable. However, subdifferentials $\partial f(\mathbf{w})$ are defined at every point (for convex functions). The subdifferential is a set of vectors called subgradients $z \in f(\mathbf{w})$ which define linear underapproximations to f , namely $f(\mathbf{w}) + z \cdot (\mathbf{w}' - \mathbf{w}) \leq f(\mathbf{w}')$ for all \mathbf{w}' .



Summary so far

$$\underbrace{\mathbf{w} \cdot \phi(x)}_{\text{score}}$$

	Classification	Regression
Predictor f_w	sign(score)	score
Relate to correct y	margin (score y)	residual (score $- y$)
Loss functions	zero-one hinge logistic	squared absolute deviation
Algorithm	SGD	SGD

Next lecture

Features:

$f_w(x)$ based on score $\mathbf{w} \cdot \phi(x)$

Which feature vector $\phi(x)$ to use?

Beyond optimization:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

How do we **generalize** beyond the training set?