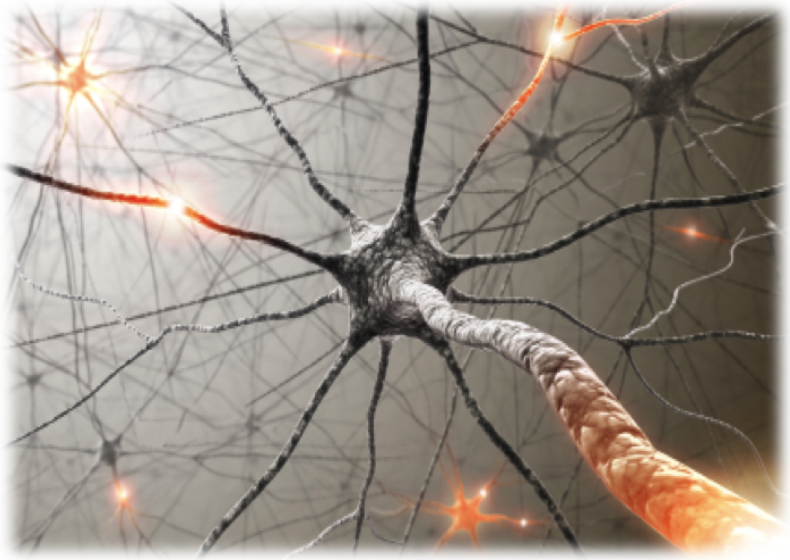
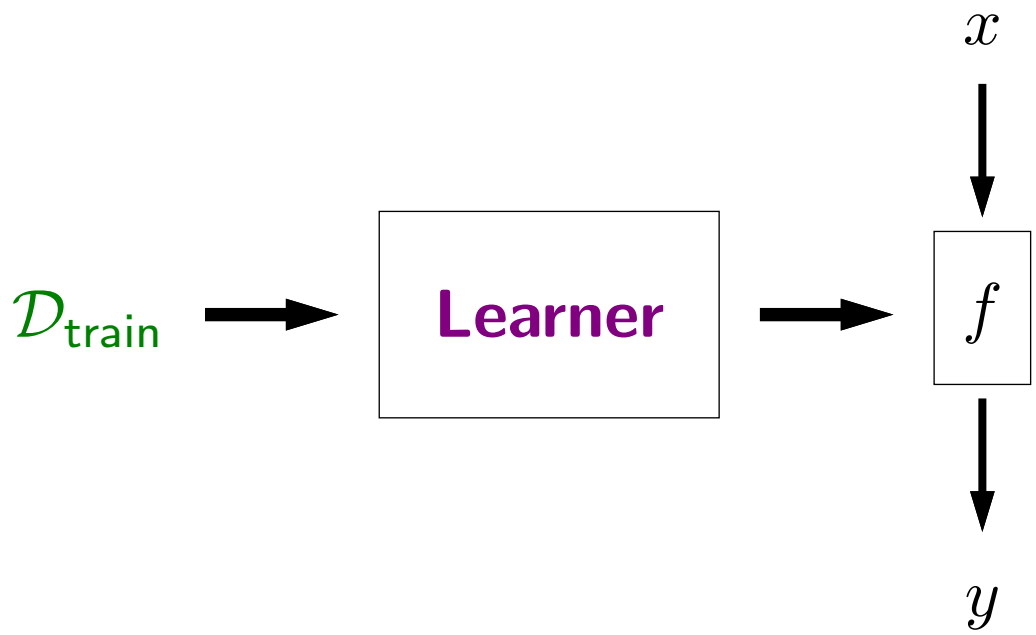
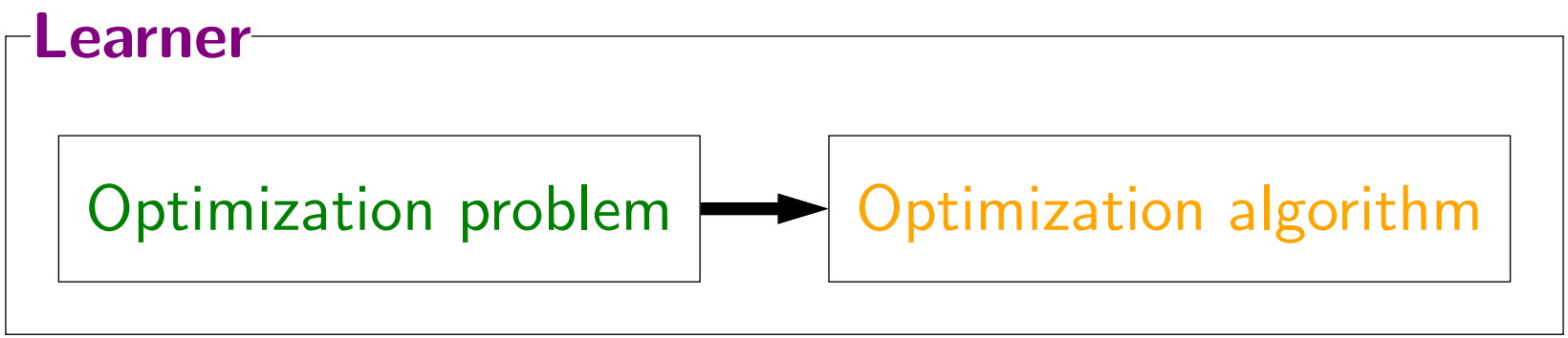
Lecture 3.1: Machine learning II



Framework


x
$\mathcal{D}_{\text{train}}$
Learner
f
y


Learner
Optimization problem
Optimization algorithm

Review

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

	Classification	Linear regression
Predictor $f_{\mathbf{w}}$	sign(score)	score
Relate to correct y	margin $=$ score y	residual $=$ score $- y$
Loss functions	zero-one	squared absolute deviation
Algorithm	?	?

- Last time, we started by studying the predictor f, concerning ourselves with linear predictors based on the score $\mathbf{w} \cdot \phi(x)$, where $\mathbf{w}$ is the weight vector we wish to learn and ϕ is the feature extractor that maps an input x to some feature vector $\phi(x) \in \mathbb{R}^d$, turning something that is domain-specific (images, text) into a mathematical object.
- Then we looked at how to learn such a predictor by formulating an optimization problem.
- Today we will close the loop by showing how to solve the optimization problem.

Review: optimization problem

Key idea: minimize training loss

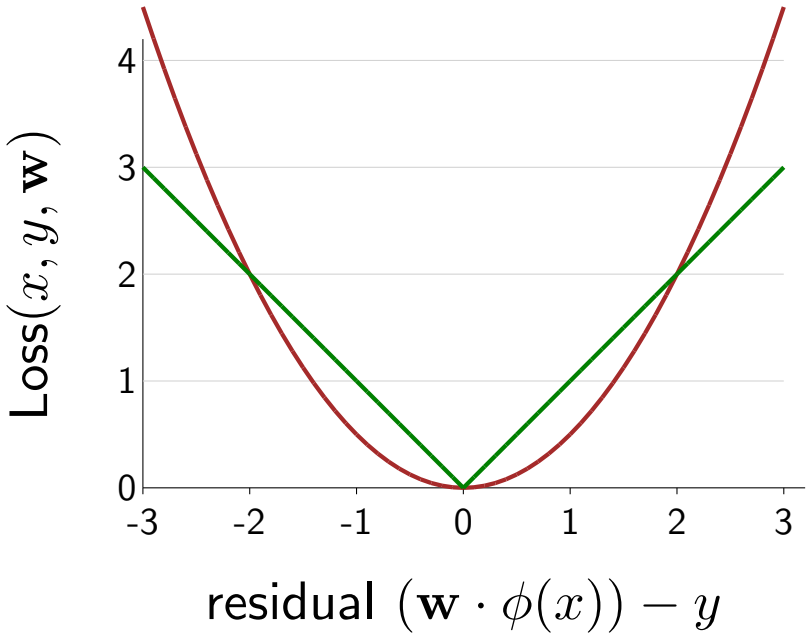
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

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

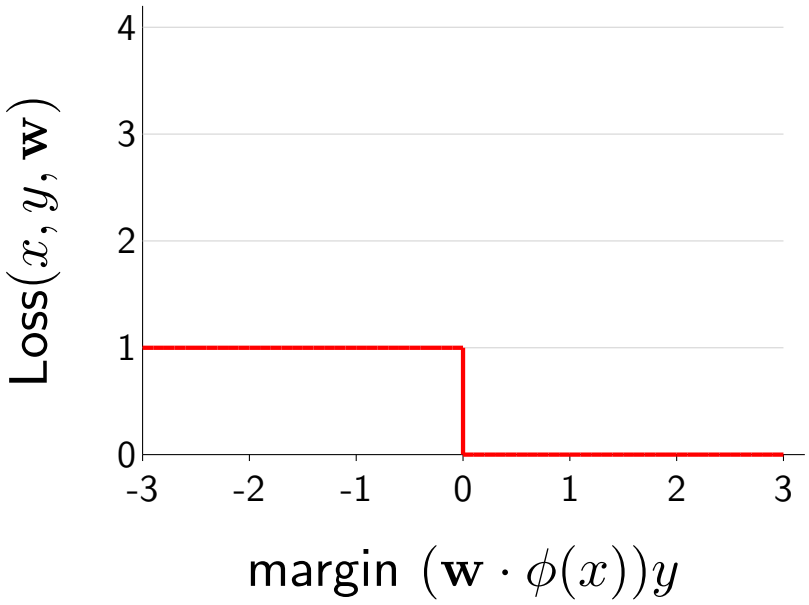
- Recall that the optimization problem was to minimize the training loss, which is the average loss over all the training examples.

Review: loss functions

Regression



Binary classification



Captures properties of the desired predictor

- The actual loss function depends on what we're trying to accomplish. Generally, the loss function takes the score $\mathbf{w} \cdot \phi(x)$, compares it with the correct output y to form either the residual (for regression) or the margin (for classification).
- Regression losses are smallest when the residual is close to zero. Classification losses are smallest when the margin is large. Which loss function we choose depends on the desired properties. For example, the absolute deviation loss for regression is robust against outliers. We will consider other losses for classification later on in the lecture.
- Note that we've been talking about the loss on a single example, and plotting it in 1D against the residual or the margin. Recall that what we're actually optimizing is the training loss, which sums over all data points. To help visualize the connection between a single loss plot and the more general picture, consider the simple example of linear regression on three data points: $([1, 0], 2)$, $([1, 0], 4)$, and $([0, 1], -1)$, where $\phi(x) = x$.
- Let's try to draw the training loss, which is a function of $\mathbf{w} = [w_1, w_2]$. Specifically, the training loss is $\frac{1}{3}((w_1 - 2)^2 + (w_1 - 4)^2 + (w_2 - (-1))^2)$. The first two points contribute a quadratic term sensitive to w_1, and the third point contributes a quadratic term sensitive to w_2. When you combine them, you get a quadratic centered at $[3, -1]$. (Draw this on the whiteboard).

Roadmap

Gradient Descent

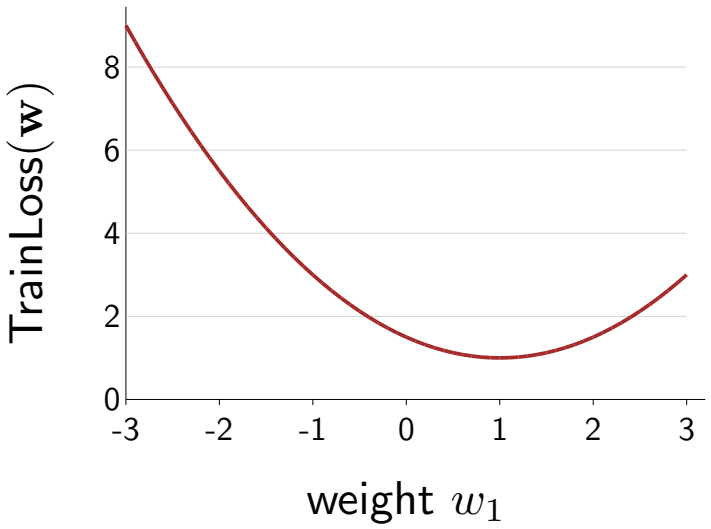
Stochastic Gradient Descent

Classification loss functions

Optimization problem

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

$\mathbf{w} \in \mathbb{R}$

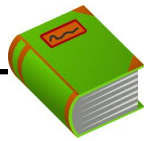


$\mathbf{w} \in \mathbb{R}^2$

[gradient plot]

- We have worked through a simple example where we can directly compute the value of $\mathbf{w}$ that minimizes TrainLoss($\mathbf{w}$). Now we will see a more general way to solve the optimization problem using gradient descent. Gradient descent is a very powerful technique that can be used to solve a wide variety of different optimization problems. For example, it can handle a variety of different loss functions.

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 $\mathbf{w} = [0, \ldots, 0]$

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

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

- 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:

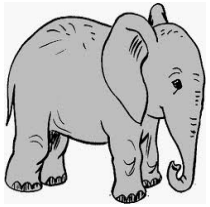
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (\mathbf{w} \cdot \phi(x) - y)^2$$

Gradient (use chain rule):

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

[live solution]

- 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 $\mathbf{w}$ only affects the loss through $\mathbf{w} \cdot \phi(x)$.



Gradient descent is slow

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

Gradient descent:

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

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?

Roadmap

Gradient Descent

Stochastic Gradient Descent

Classification loss functions

Stochastic gradient descent

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

Gradient descent (GD):

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

Stochastic gradient descent (SGD):

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

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

Key idea: stochastic updates

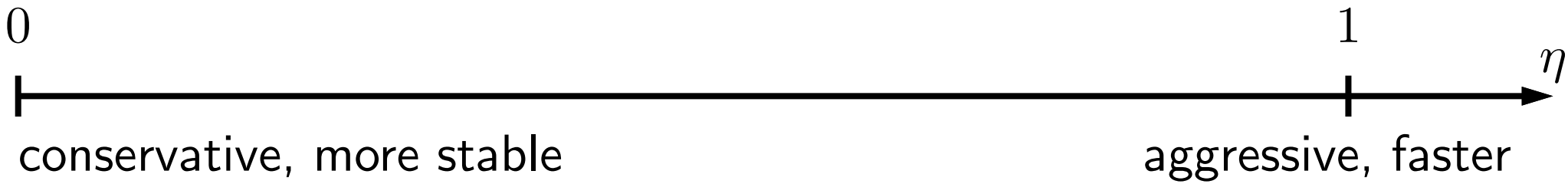
It's not about **quality**, it's about **quantity**.

- 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 $\mathbf{w}$ 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 have all the positive examples first and the negative examples after that.

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

- 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).

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?

Roadmap

Gradient Descent

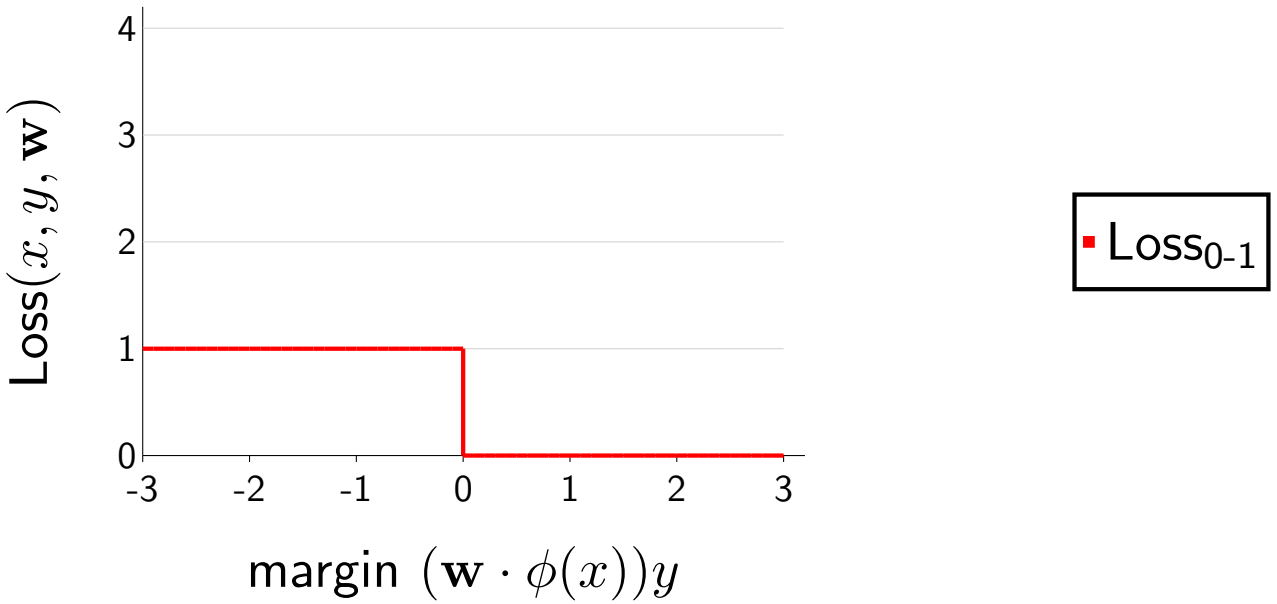
Stochastic Gradient Descent

Classification loss functions

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

Zero-one loss

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



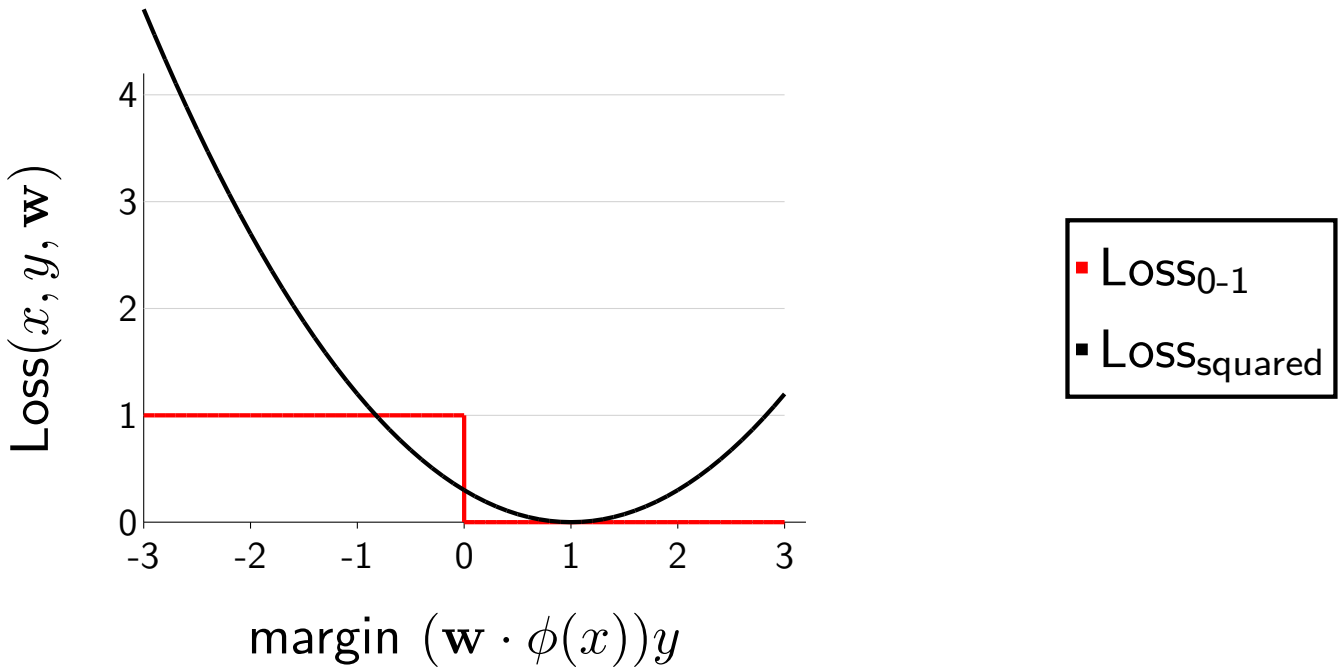
Problems:

- Gradient of Loss$_{0\text{-}1}$ is 0 everywhere, SGD not applicable
- Loss$_{0\text{-}1}$ is insensitive to how badly the model messed up

- 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.

Squared loss?

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



Good: Differentiable

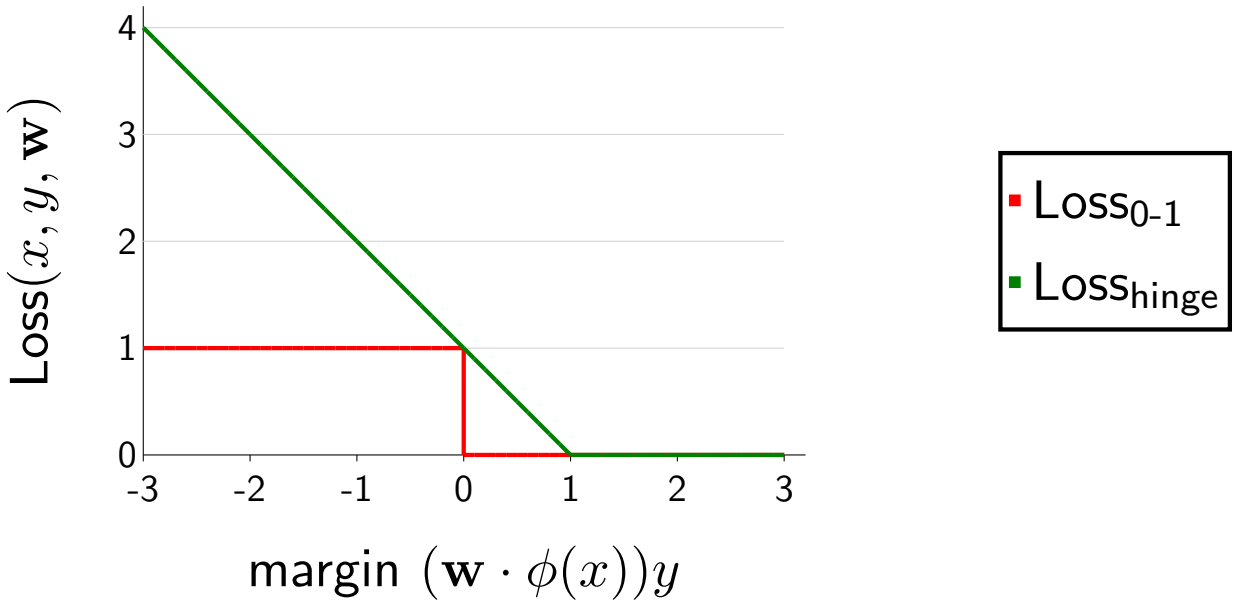
Bad: Penalizes large margins

Overall: Not suitable for classification

- It might be tempting to use squared loss for classification, since we have already seen that it's useful for regression.
- However, that doesn't optimize for what we want in a classifier. Recall that for classification, the margin measures how confident the classifier is in the correct label. We want the margin on each example to be as high as possible. Squared loss will encourage the margin to be exactly 1 on all examples, and penalize margins that are higher.
- Geometric intuition also tells us why this is a bad choice of loss. Remember the picture from last class, where we drew the decision boundary for classification. The margin measures how far the point is from the decision boundary. It is not reasonable to expect all margins to be exactly 1–there may be some points that are far away from the decision boundary and are easy to classify. Using squared loss in such a case would punish the correct weight vector.

Support vector machines

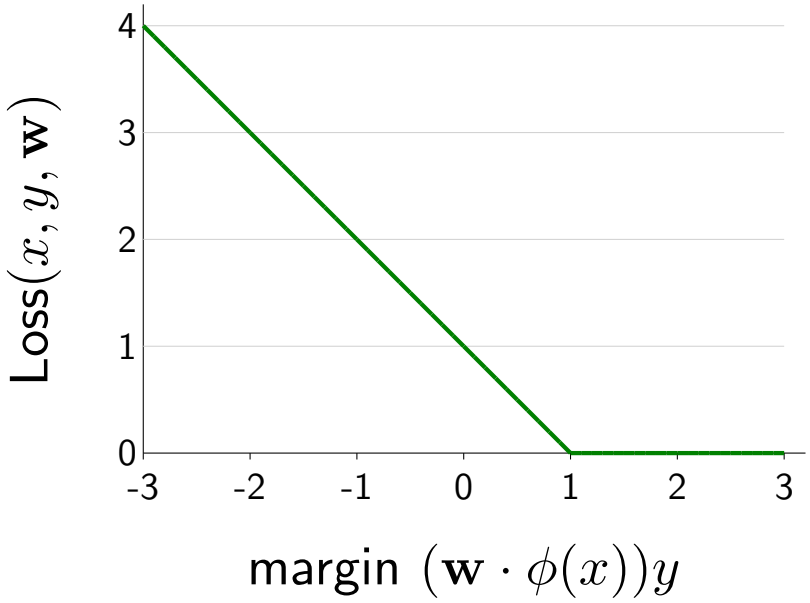
$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$



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

- Instead of using the squared loss, let's create a loss function that is both differentiable and monotonically decreasing in the margin (i.e., larger margin can never lead to larger loss). One option is 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.

A gradient exercise



Problem: Gradient of hinge loss

Compute the gradient of

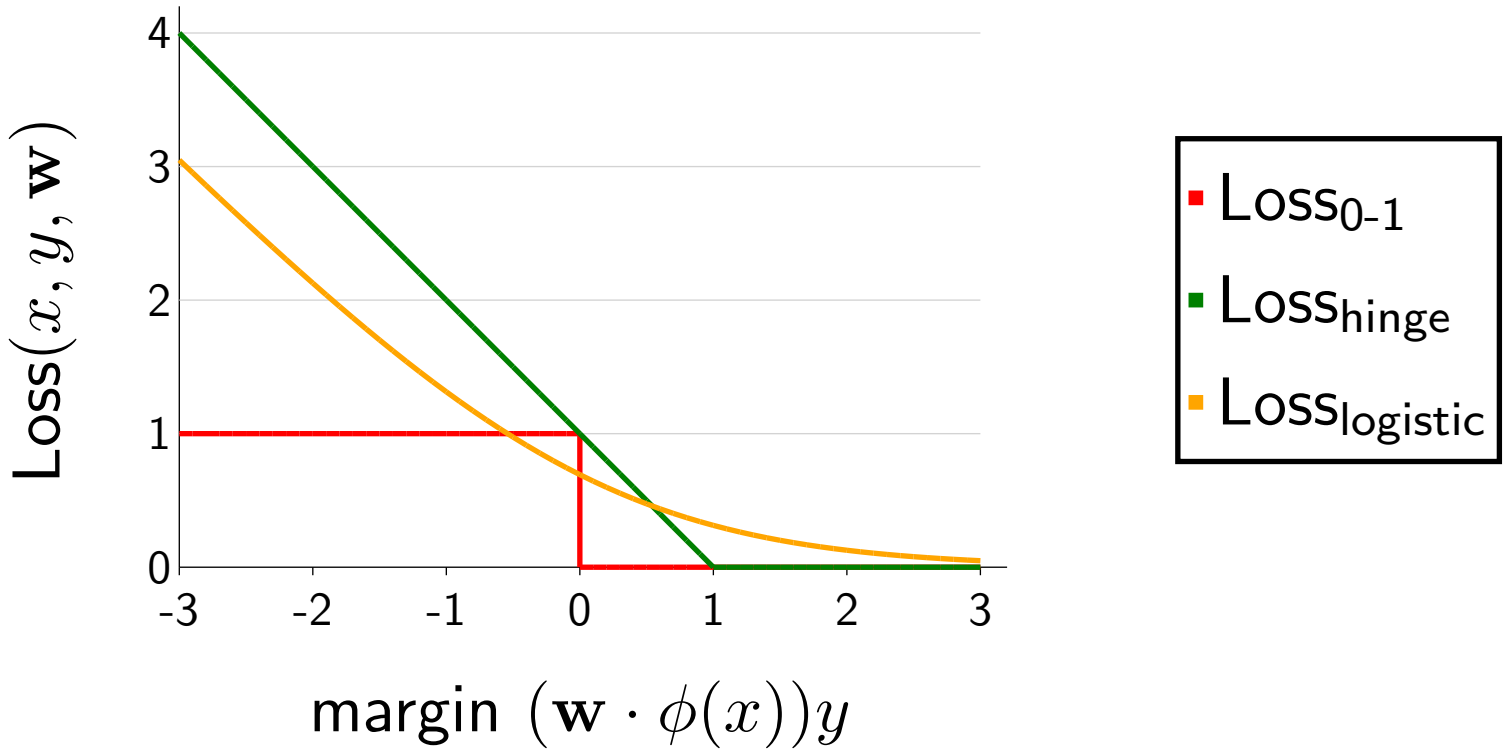
$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

[whiteboard]

- 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 $\mathbf{w}'$.

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

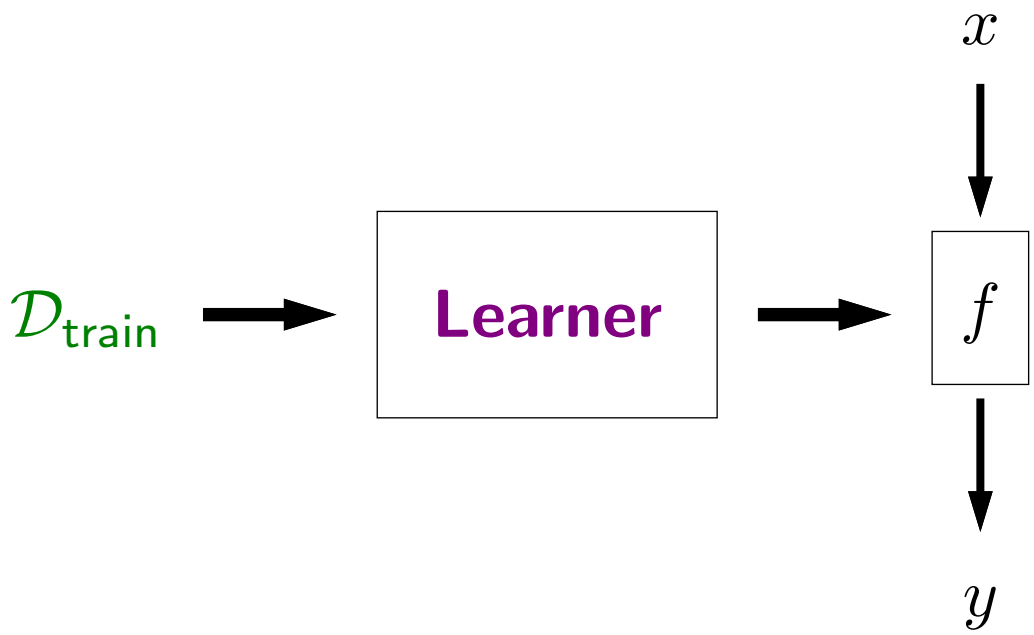
- Another popular loss function used in machine learning is the **logistic loss**. The main property of the logistic loss is no matter how correct your prediction is, you will have non-zero loss, and so there is still an incentive (although a diminishing one) to push the margin even larger. This means that you'll update on every single example.
- There are some connections between logistic regression and probabilistic models, which we will get to later.

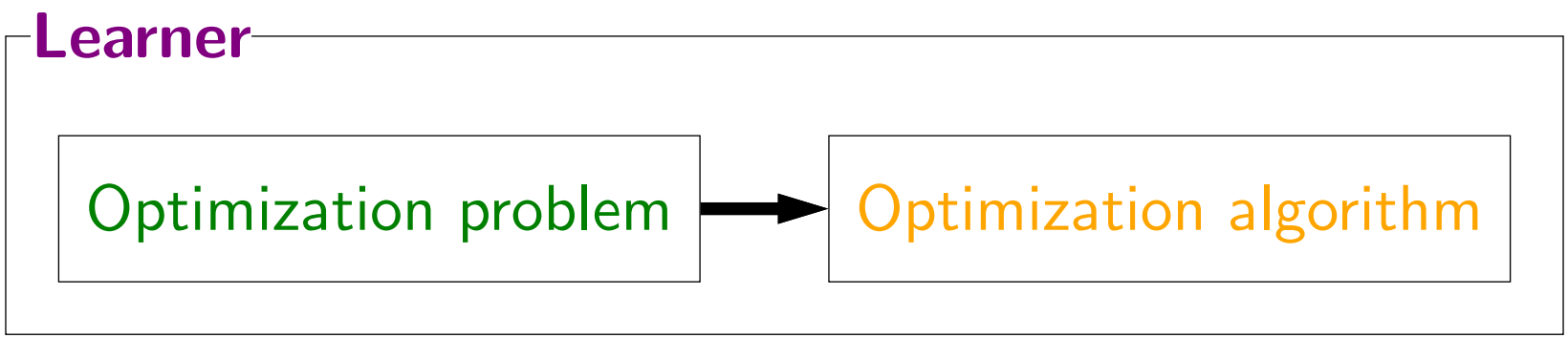
Summary

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

	Classification	Linear regression
Predictor $f_{\mathbf{w}}$	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**

Framework


x
$\mathcal{D}_{\text{train}}$
Learner
f
y


Learner
Optimization problem
Optimization algorithm

Next lecture

Linear predictors:

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

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

Loss minimization:

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

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