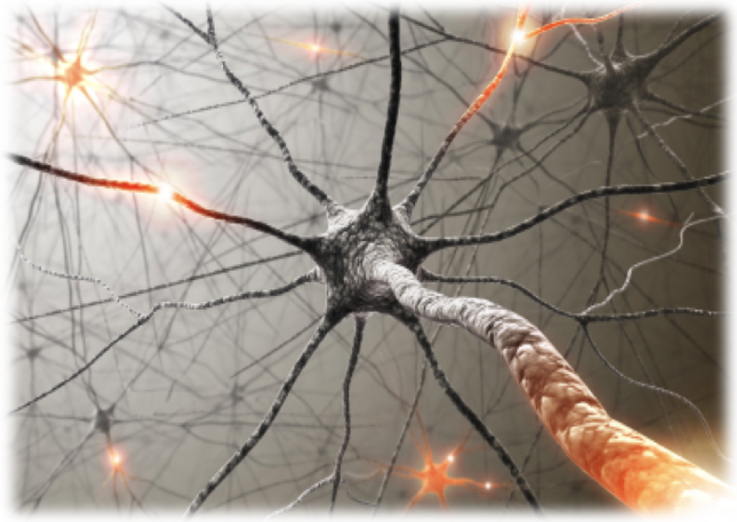
Machine learning: overview



- In this module, I will provide an overview of the topics we plan to cover under machine learning.

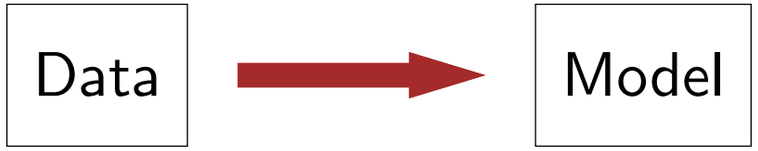
What is this animal?

- How can we build system that detects Zebras?

- Very hard to write down some simple rules

- As a motivation, lets go back 2 days You want to make a zebra detector - how are you going to do this? Writing a set of rules that looks at some pixels and makes a prediction seems insanely hard

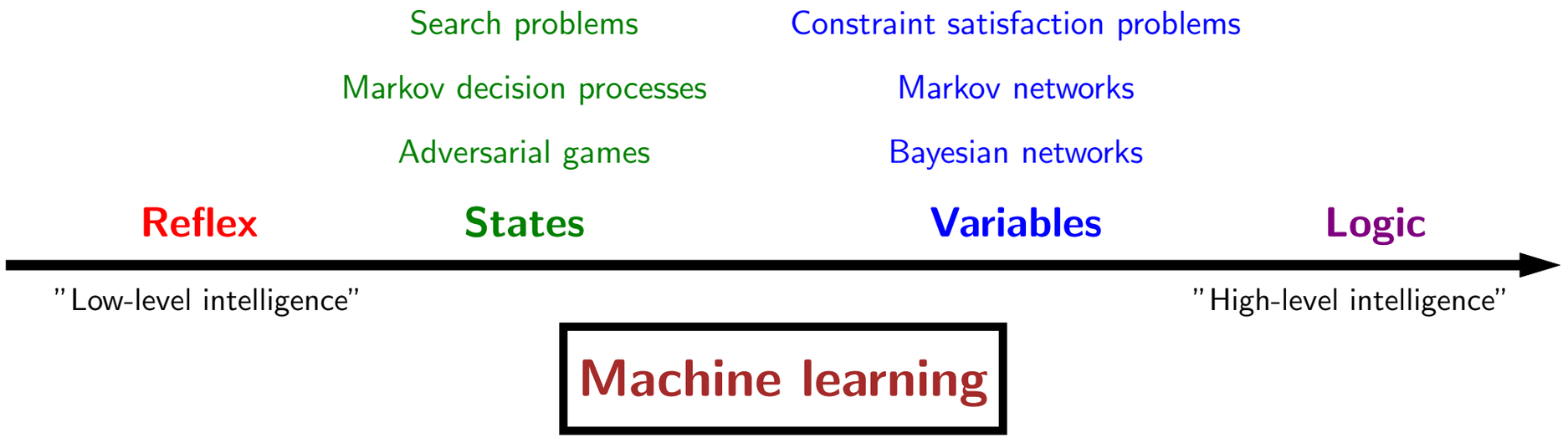
Machine learning



- Key observation: easy to collect zebras vs non-zebra images

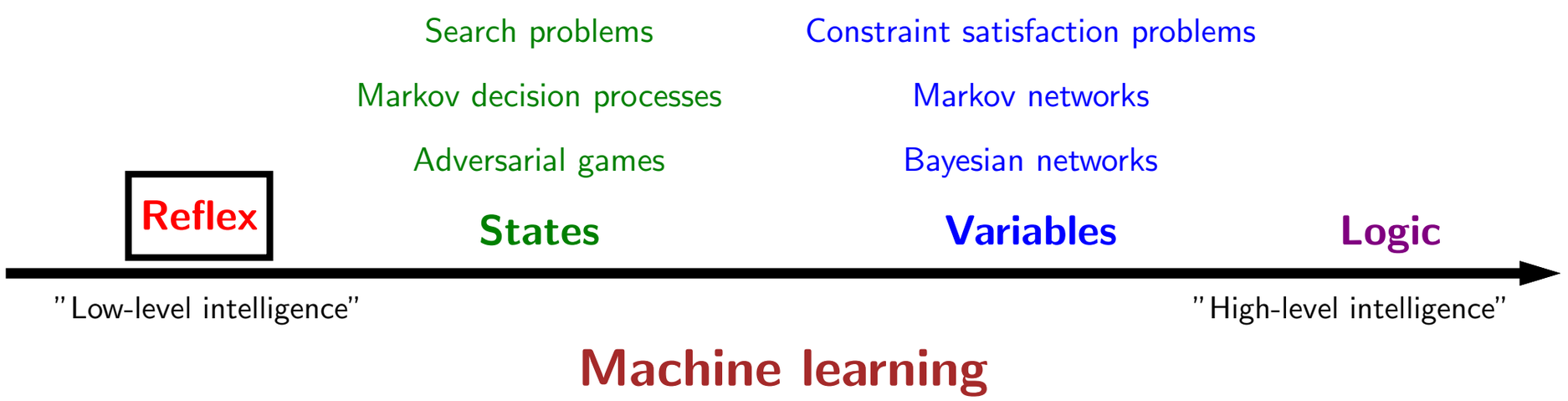
- Just have a model 'learn' how to detect zebras

Course plan


Search problems
Constraint satisfaction problems
Markov decision processes
Markov networks
Adversarial games
Bayesian networks
Reflex
States
Variables
Logic
"Low-level intelligence"
"High-level intelligence"
Machine learning

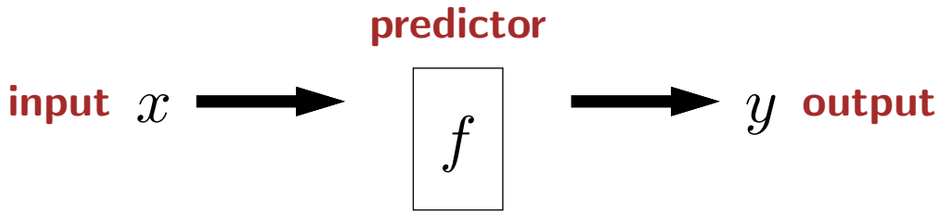
- Recall that this process of machine learning turns data into a model. Then with that model, you can perform inference on it to make predictions.

Course plan


Search problems
Markov decision processes
Adversarial games
Constraint satisfaction problems
Markov networks
Bayesian networks
Reflex
States
Variables
Logic
"Low-level intelligence"
"High-level intelligence"
Machine learning

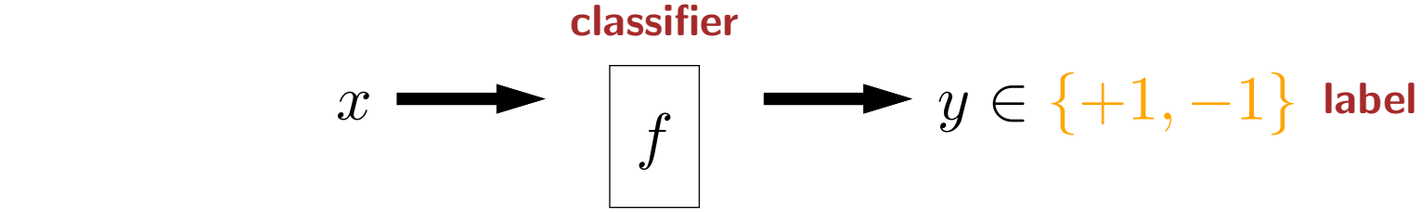
- While machine learning can be applied to any type of model, we will focus our attention on reflex-based models, which include models such as linear classifiers and neural networks.
- In reflex-based models, inference (prediction) involves a fixed set of fast, feedforward operations.

Reflex-based models



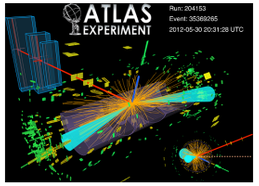
- Abstractly, a **reflex-based model** (which we will call a **predictor** f) takes some **input** x and produces some **output** y.
- (In statistics, y is known as the response, and when x is a real vector, it is known as covariates, features or sometimes predictors, which is an unfortunate naming clash.)
- The input can usually be arbitrary (an image or sentence), but the form of the output y is generally restricted, and what it is determines the type of **prediction task**.

Binary classification



Fraud detection: credit card transaction $\rightarrow$ fraud or no fraud

Toxic comments: online comment $\rightarrow$ toxic or not toxic

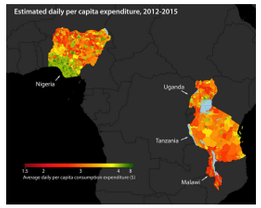


Higgs boson: measurements of event $\rightarrow$ decay event or background

- One common prediction task is binary classification, where the output y, typically expressed as positive (+1) or negative (-1).
- In the context of classification tasks, f is called a **classifier** and y is called a **label** (sometimes class, category, or tag).
- Here are some practical applications.
- One application is fraud detection: given information about a credit card transaction, predict whether it is a fradulent transaction or not, so that the transaction can be blocked.
- Another application is moderating online discussion forums: given an online comment, predict whether it is toxic (and therefore should get flagged or taken down) or not.
- A final application comes from physics: After the discovery of the Higgs boson, scientists were interested in how it decays. The Large Hadron Collider at CERN smashes protons against each other and then detects the ensuing events. The goal is to predict whether each event is a Higgs boson decaying (into two tau particles) or just background noise.
- Each of these applications has an associated Kaggle dataset. You can click on the pictures to find out more details.
- As an aside, **multiclass classification** is a generalization of binary classification where the output y could be one of K possible values. For example, in digit classification, $K = 10$.

Regression

$$x \longrightarrow \boxed{f} \longrightarrow y \in \mathbb{R}$$ **response**



Poverty mapping: satellite image $\rightarrow$ asset wealth index

Housing: information about house $\rightarrow$ price

Arrival times: destination, weather, time $\rightarrow$ time of arrival

- The second major type of prediction task we'll cover is regression. Here, the output y is a real number (often called the **response** or target).
- One application is poverty mapping: given a satellite image, predict the average asset wealth index of the homes in that area. This is used to measure poverty across the world and determine which areas are in greatest need of aid.
- Another application: given information about a house (e.g., location, number of bedrooms), predict its price.
- A third application is to predict the arrival time of some service, which could be package deliveries, flights, or rideshares.
- The key distinction between classification and regression is that classification has **discrete** outputs (e.g., "yes" or "no" for binary classification), whereas regression has **continuous** outputs.

Structured prediction

$x \longrightarrow \boxed{f} \longrightarrow y$ is a complex object

Machine translation: English sentence $\rightarrow$ Japanese sentence

Dialogue: conversational history $\rightarrow$ next utterance

Image captioning: image $\rightarrow$ sentence describing image

Image segmentation: image $\rightarrow$ segmentation

- The final type of prediction task we will consider is structured prediction, which is a bit of a catch all.
- In **structured prediction**, the output y is a complex object, which could be a sentence or an image. So the space of possible outputs is huge.
- One application is machine translation: given an input sentence in one language, predict its translation into another language.
- Dialogue can be cast as structured prediction: given the past conversational history between a user and an agent (in the case of virtual assistants), predict the next utterance (what the agent should say).
- In image captioning, say for visual assistive technologies: given an image, predict a sentence describing what is in that image.
- In image segmentation, which is needed to localize objects for autonomous driving: given an image of a scene, predict the segmentation of that image into regions corresponding to objects in the world.
- Generating an image or a sentence can seem daunting, but there's a secret here. A structured prediction task can often be broken up into a sequence of multiclass classification tasks. For example, to predict an entire sentence, predict one word at a time, going left to right. This is a very powerful reduction!
- Unfortunately, there is no free lunch here. The reduction is computationally easier but just as difficult as the original problem. You can see this intuitively by noticing that if we are predicting words one at a time errors can quickly cascade, making early errors result in more errors in the future

Roadmap

Tasks

Linear regression

Linear classification

K-means

Models

Non-linear features

Feature templates

Neural networks

Differentiable programming

Algorithms

Stochastic gradient descent

Backpropagation

Considerations

Generalization

Best practices

- Here are the rest of the modules under the machine learning unit.
- In the first week, we will start with the two most basic and fundamental settings in machine learning. Linear regression and classification, which assumes that the relation between inputs and outputs are linear and we will discuss a way to train these models using gradient descent
- Next, we will introduce **stochastic gradient descent**, and show that it can be much faster than vanilla gradient descent.
- In the second week, we will go beyond linear models and show how you can define **non-linear features**, which effectively gives us non-linear predictors using the machinery of linear models! **Feature templates** provide us with a framework for organizing the set of features.
- Then we introduce **neural networks**, which also provide non-linear predictors, but allow these non-linearities to be learned automatically from data. We follow up immediately with **backpropagation**, an algorithm that allows us to automatically compute gradients that we need to train these models.
- We then briefly discuss the extension of neural networks to **differentiable programming**, which allows us to easily build up many of the existing state-of-the-art deep learning models in NLP and computer vision like lego blocks.
- So far we have focused on supervised learning where we have paired inputs and ouputs. We take a brief detour and discuss **K-means**, which is a simple unsupervised learning algorithm for clustering data points when we have no output labels.
- Having covered these, we will end trying to gain a deeper understanding of machine learning **Generalization** is about understanding why it is possible to learn these models at all: when does a model trained on set of training examples actually generalize to new test inputs? This is where model complexity comes up. Finally, we discuss **best practices** for doing machine learning in practice.

Roadmap

Topics in the lecture:

Linear regression

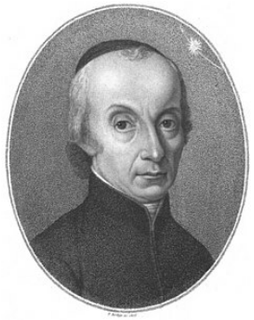
Linear classification

Stochastic gradient descent

Losses over groups (group DRO)

- We will be covering linear regression, classification, and stochastic gradient descent today..
- Lets dive right into learning about linear regression

The discovery of Ceres



1801: astronomer Piazzi discovered Ceres, made 19 observations of location before it was obscured by the sun

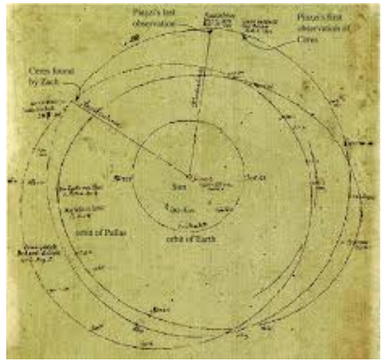
Time	**Right ascension**	**Declination**
Jan 01, 20:43:17.8	50.91	15.24
Jan 02, 20:39:04.6	50.84	15.30
...	...	...
Feb 11, 18:11:58.2	53.51	18.43

When and where will Ceres be observed again?

- Our story of linear regression starts on January 1, 1801, when an Italian astronomer Giuseppe Piazzi noticed something in the night sky while looking for stars, which he named Ceres. Was it a comet or a planet? He wasn't sure.
- He observed Ceres over 42 days and wrote down 19 data points, where each one consisted of a timestamp along with the right ascension and declination, which identifies the location in the sky.
- Then Ceres moved too close to the sun and was obscured by its glare. Now the big question was when and where will Ceres come out again?
- It was now a race for the top astronomers at the time to answer this question.

Gauss's triumph

September 1801: Gauss took Piazzi's data and created a model of Ceres's orbit, makes prediction

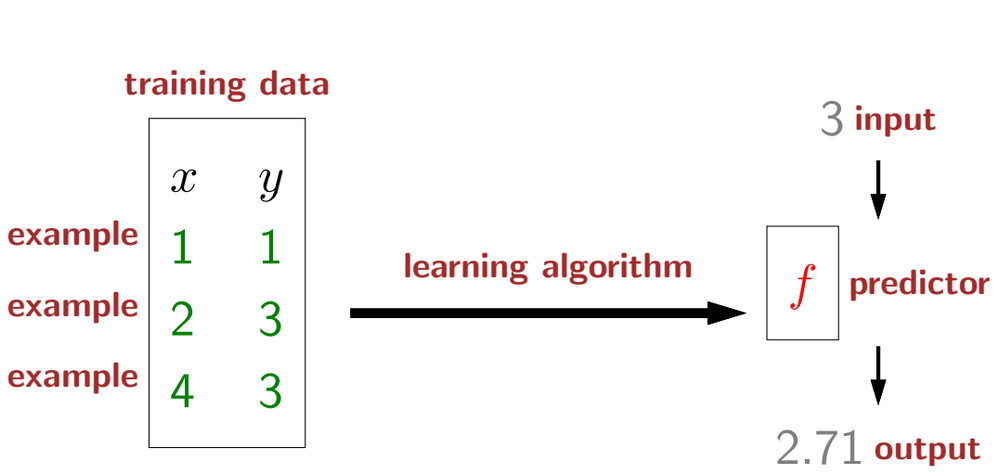


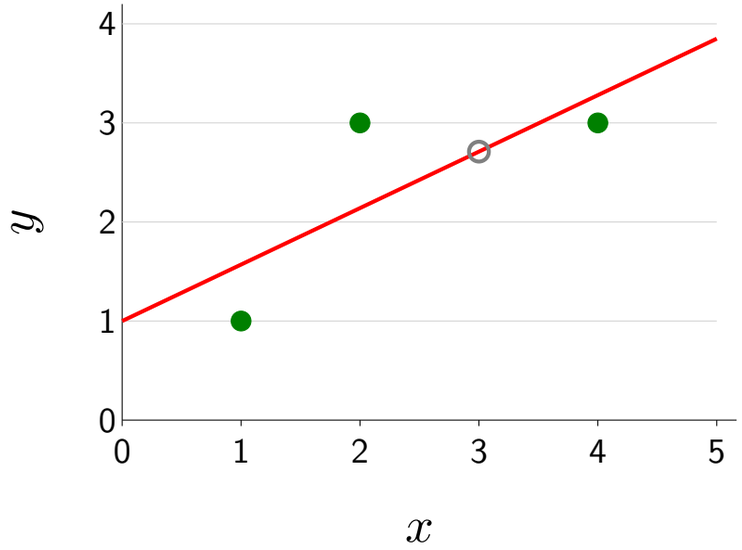
December 7, 1801: Ceres located within 1/2 degree of Gauss's prediction, much more accurate than other astronomers

Method: least squares linear regression

- Carl Friedrich Gauss, the famous German mathematician, took the data and developed a model of Ceres's orbit and used it to make a prediction.
- Clearly without a computer, Gauss did all his calculations by hand, taking over 100 hours.
- This prediction was actually quite different than the predictions made by other astronomers, but in December, Ceres was located again, and Gauss's prediction was by far the most accurate.
- Gauss was very secretive about his methods, and a French mathematician Legendre actually published the same method in 1805, though Gauss had developed the method as early as 1795.
- The method here is least squares linear regression, which is a simple but powerful method used widely today, and it captures many of the key aspects of more advanced machine learning techniques.

Linear regression framework





Design decisions:

- Which predictors are possible? **hypothesis class**
- How good is a predictor? **loss function**
- How do we compute the best predictor? **optimization algorithm**

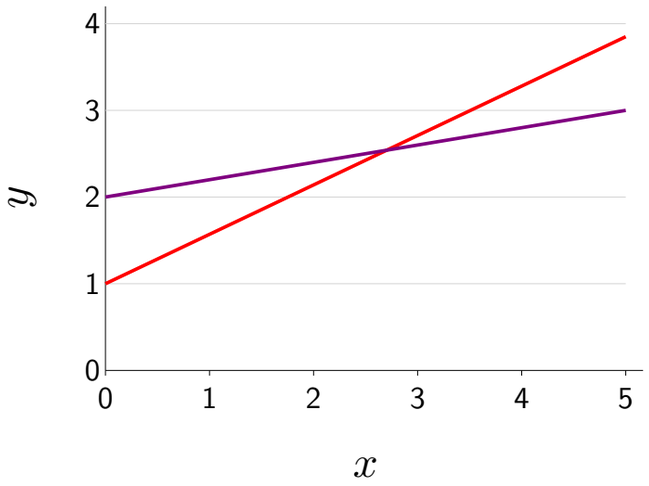
- Let us now present the linear regression framework.
- Suppose we are given **training data**, which consists of a set of examples. Each **example** (also known as data point, instance, case) consists of an input x and an output y. We can visualize the training set by plotting y against x.
- A learning algorithm takes the training data and produces a model f, which we will call a **predictor** in the context of regression. In this example, f is the red line.
- This predictor allows us to make predictions on new inputs. For example, if you feed 3 in, you get $f(3)$, corresponding to the gray circle.
- There are three design decisions to make to fully specify the learning algorithm:
- First, which predictors f is the learning algorithm allowed to produce? Do we draw a straight line, or can we draw a smooth line that passes through all the points? This choice is what we call the **hypothesis class**, since it is a hypothesis about possible relationships between x and y
- Second, how does the learning algorithm judge which predictor is good? In this case, you and I are probably eyeballing the distance between the point and the line but we need to express this more formally. This is the **loss function** which measures how well a predictor fits an example
- Finally, how can we find the lowest loss predictor in our hypothesis class? Here, we might as for some explicit algorithm that gives us the line that minimizes the vertical distance to the points. We call this the **optimization algorithm**?

Hypothesis class: which predictors?

$$f(x) = 1 + 0.57x$$

$$f(x) = 2 + 0.2x$$

$$f(x) = w_1 + w_2 x$$



Vector notation:

weight vector $\mathbf{w} = [w_1, w_2]$ feature extractor $\phi(x) = [1, x]$ feature vector

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

$$f_{\mathbf{w}}(3) = [1, 0.57] \cdot [1, 3] = 2.71$$

Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^2\}$$

- Let's consider the first design decision: what is the hypothesis class? One possible predictor is the red line, where the intercept is 1 and the slope is 0.57, Another predictor is the purple line, where the intercept is 2 and the slope is 0.2.
- In general, let's consider all predictors of the form $f(x) = w_1 + w_2 x$, where the intercept w_1 and and the slope w_2 can be arbitrary real numbers.
- We are going to write this down in a slightly more general, vector notation which will be handy later. Let's first pack the intercept and slope into a single vector, which we will call the **weight vector** w (more generally we are going to call this the parameters of the model).
- Similarly, we will define a **feature extractor** (also called a feature map) ϕ, which takes x and converts it into the **feature vector** with two elements. The first element is a constant one, and the second is the value of x itself
- Now we can succinctly write the predictor $f_{\mathbf{w}}$ to be the dot product between the weight vector and the feature vector, which we call the **score**.
- To see this predictor in action, let us feed $x = 3$ as the input and take the dot product.
- We now have a way of writing down predictors We can use this to define the hypothesis class $\mathcal{F}$ as the set of all possible predictors $f_{\mathbf{w}}$ as we range over all possible weight vectors $\mathbf{w}$. Our goal will be to find the best predictor within this class of linear functions

Loss function: how good is a predictor?

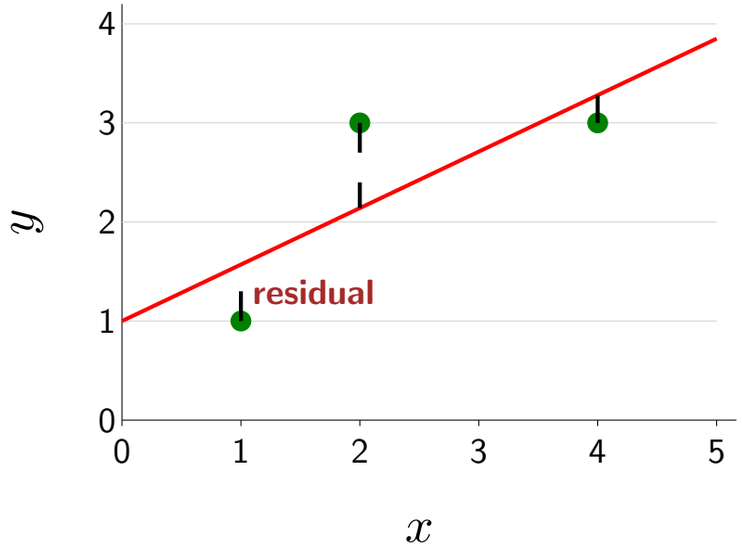
$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

$\mathbf{w} = [1, 0.57]$

$\phi(x) = [1, x]$

training data $\mathcal{D}_{\text{train}}$

x	y
1	1
2	3
4	3



$$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$$ **squared loss**

$$\text{Loss}(1, 1, [1, 0.57]) = ([1, 0.57] \cdot [1, 1] - 1)^2 = 0.32$$

$$\text{Loss}(2, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 2] - 3)^2 = 0.74$$

$$\text{Loss}(4, 3, [1, 0.57]) = ([1, 0.57] \cdot [1, 4] - 3)^2 = 0.08$$

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

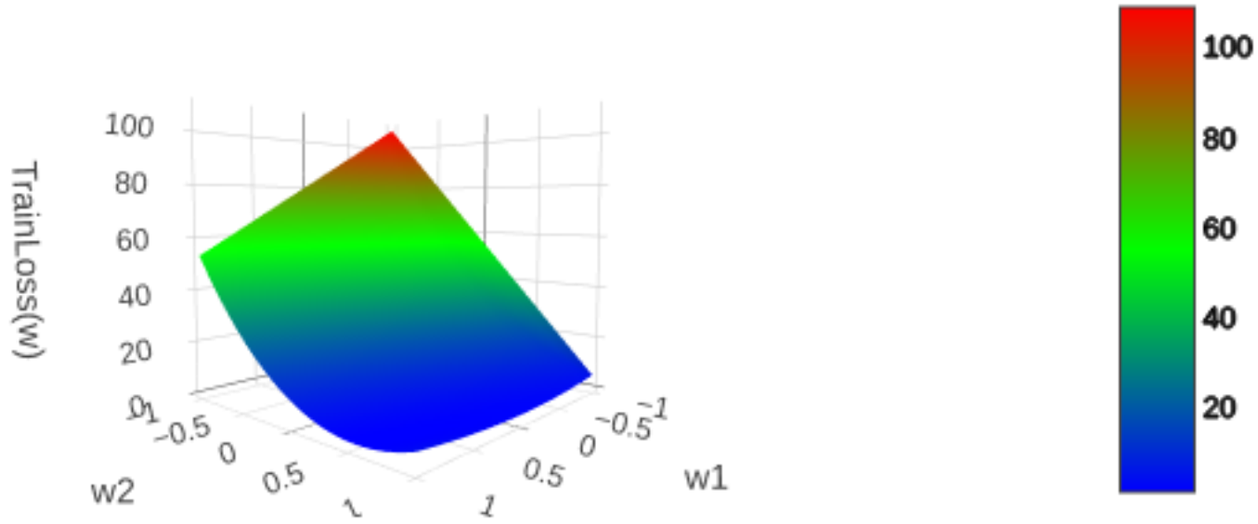
$$\text{TrainLoss}([1, 0.57]) = 0.38$$

- The next design decision is how to define what we mean when we say a predictor is good or bad
- Going back to our running example, let's consider the red predictor defined by the weight vector [1, 0.57], and the three training examples.
- Intuitively, a predictor is good if it can fit the training data. For each training example, let us look at the difference between the predicted output $f_{\mathbf{w}}(x)$ and the actual output y, known as the **residual**.
- Now define the **loss function** on given example with respect to $\mathbf{w}$ to be the residual squared (giving rise to the term least squares). This measures how badly the function f screwed up on that example.
- If this is your first time seeing regression, you might be wondering why we are squaring the residual. There are somewhat deeper reasons from statistical and optimization perspectives for why squaring would be the natural thing to do I will discuss this briefly later, but if you are interested, I encourage you to ask in office hours or Q and A.
- Returning to our loss, we now have a per-example loss computed by plugging in the example and the weight vector. Now, define the **training loss** (also known as the training error or empirical risk) to be simply the average of the per-example losses of the training examples.
- The training loss of this red weight vector is 0.38.
- If you were to plug in the purple weight vector or any other weight vector, you would get some other training loss.

Loss function: visualization

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

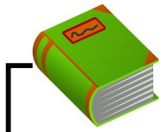
$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$



- We can visualize the training loss in this case because the weight vector $\mathbf{w}$ is only two-dimensional. In this plot, for each w_1 and w_2, we have the training loss. Red is higher, blue is lower.
- It is now clear that the best predictor is simply the one with the lowest training loss, which is somewhere down in the blue region. Formally, we are looking for the predictor that minimizes the training error, or empirical risk..

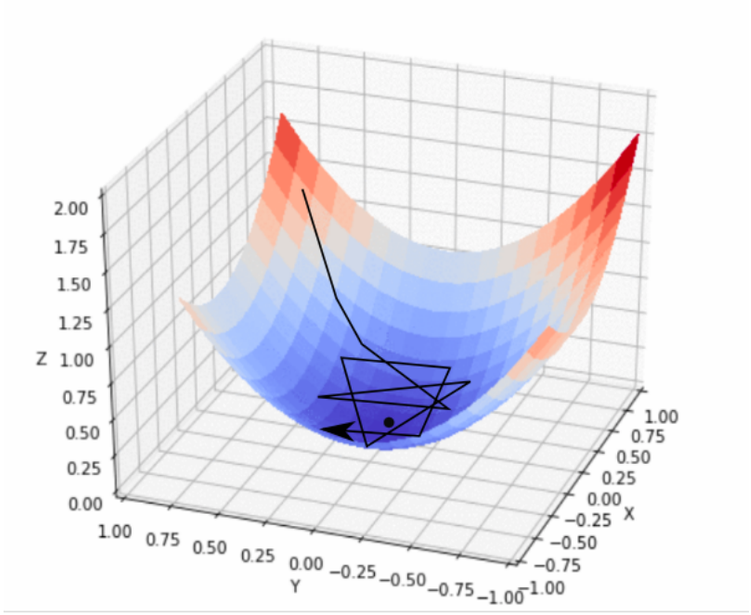
Optimization algorithm: how to compute best?

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



Definition: gradient

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



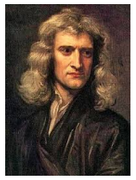
Algorithm: gradient descent

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

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

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

- Now the third design decision: how do we compute the best predictor, i.e., the solution to the optimization problem?
- The dominant approach in machine learning is to abstract away the optimization problem. Often, we can ignore the fact that we are doing linear least squares, or even machine learning and simply treat the objective function TrainLoss($\mathbf{w}$) as a black-box to optimize.
- In this case, an effective approach is **iterative optimization**, which resembles a form of trial-and-error. We start with some $\mathbf{w}$ and change it slightly to make the objective function smaller.
- But how should we be changing the function? trying random small changes to $\mathbf{w}$ is wildly inefficient Ideally, we can find the direction to modify $\mathbf{w}$ so that it makes the loss go down as fast as possible It turns out that we can identify this **exactly** since it is the negative of the gradient of the loss
- Formally, this iterative optimization procedure is called **gradient descent**. We first initialize $\mathbf{w}$ to some value.
- Then perform the following update T times, where T is called the number of **epochs**: Take the current weight vector $\mathbf{w}$ and subtract a positive constant η times the gradient. The **step size** η specifies how aggressively we want to pursue a direction. The step size η, initialization, and the number of epochs T are **hyperparameters** of the optimization algorithm, which we will discuss later.
- Finally, we are going to be discussing gradient descent because of its ease of use and popularity but gradient descent is not necessarily the best optimizer, especially for simple problems like linear regression. In our case, we can **exactly** solve the least-squares regression problem using either singular value decomposition or a pseudoinverse.



Computing the gradient

Objective function:

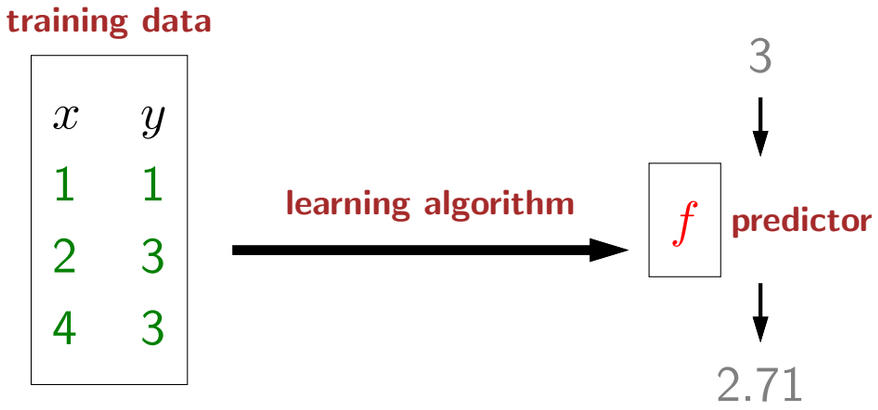
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (\mathbf{w} \cdot \phi(x) - y)^2$$

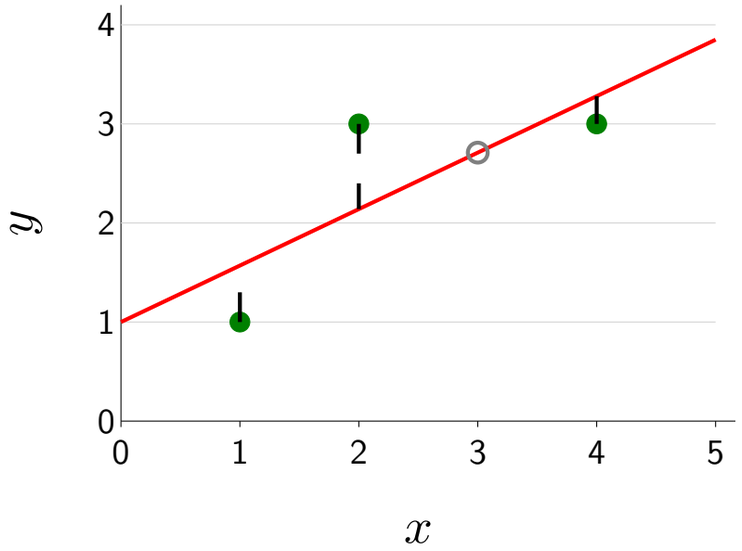
Gradient (use chain rule):

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

- To apply gradient descent, we need to compute the gradient of our objective function TrainLoss($\mathbf{w}$).
- You could throw it into TensorFlow or PyTorch, but it is pedagogically useful to do the calculus, which can be done by hand here.
- The main thing here is to remember that we're taking the gradient with respect to $\mathbf{w}$, so everything else is a constant.
- Gradients are like derivatives, so it is linear and the gradient of a sum is the sum of the gradient. We can also apply the chain rule, which means that we first take the gradient of the square, and then gradient inside the square Finally, the gradient of the dot product $\mathbf{w} \cdot \phi(x)$ is simply $\phi(x)$.
- Note that the gradient has a nice interpretation here. For the squared loss, it is the residual (prediction - target) times the feature vector $\phi(x)$.
- This is a very natural update function: take the prediction error and try to map the error back to the weights. Going back to our earlier discussion about loss function, this is related to the optimization reason for choosing the squared loss, when a model makes very small error, its gradient is also small. The same is not true of using the absolute value.

Summary





Which predictors are possible?

Hypothesis class

Linear functions

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)\}, \phi(x) = [1, x]$

How good is a predictor?

Loss function

Squared loss

$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$

How to compute best predictor?

Optimization algorithm

Gradient descent

$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \textbf{TrainLoss}(\mathbf{w})$

- In this module, we have gone through the basics of linear regression. A learning algorithm takes training data and produces a predictor f, which can then be used to make predictions on new inputs.
- Then we addressed the three design decisions:
- First, what is the hypothesis class (the space of allowed predictors)? We focused on linear functions, but we will later see how this can be generalized to other feature extractors to yield non-linear functions, and beyond that, neural networks.
- Second, how do we assess how good a given predictor is with respect to the training data? For this we used the squared loss, which gives us least squares regression. We will see later how other losses allow us to handle problems such as classification.
- Third, how do we compute the best predictor? We described the simplest procedure, gradient descent. Later, we will see how stochastic gradient descent can be much more computational efficient.
- And that concludes this module.

Roadmap

Topics in the lecture:

Linear regression

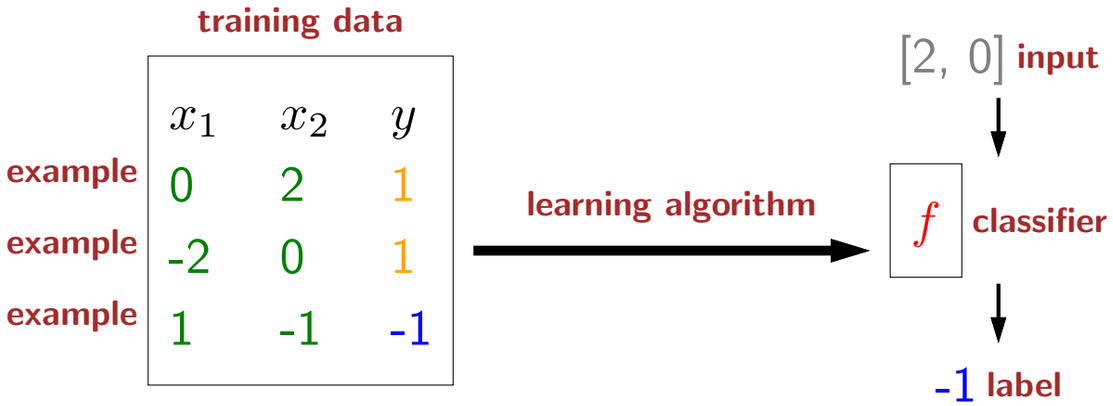
Linear classification

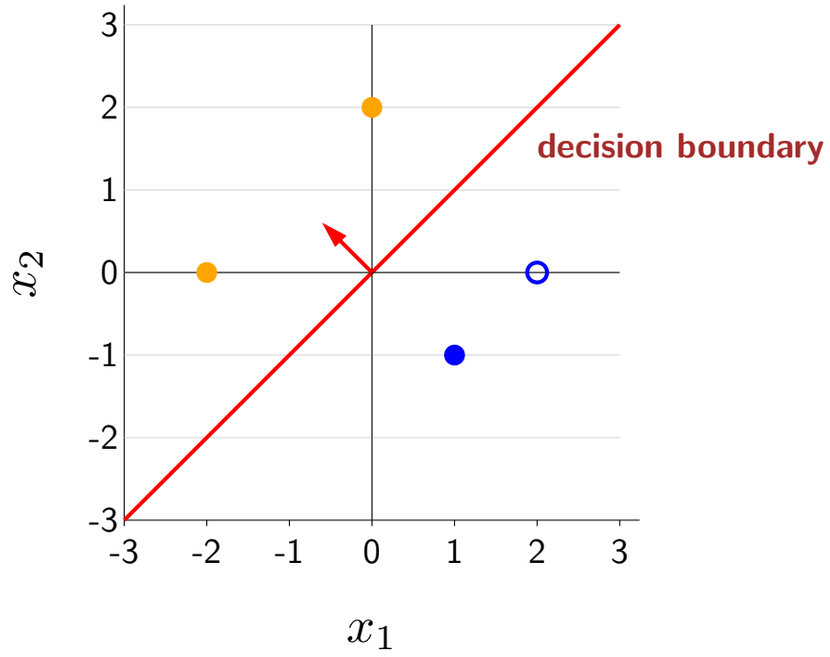
Stochastic gradient descent

Losses over groups (group DRO)

- We now present linear (binary) classification, working through a simple example just like we did for linear regression.

Linear classification framework





Design decisions:

Which classifiers are possible? **hypothesis class**

How good is a classifier? **loss function**

How do we compute the best classifier? **optimization algorithm**

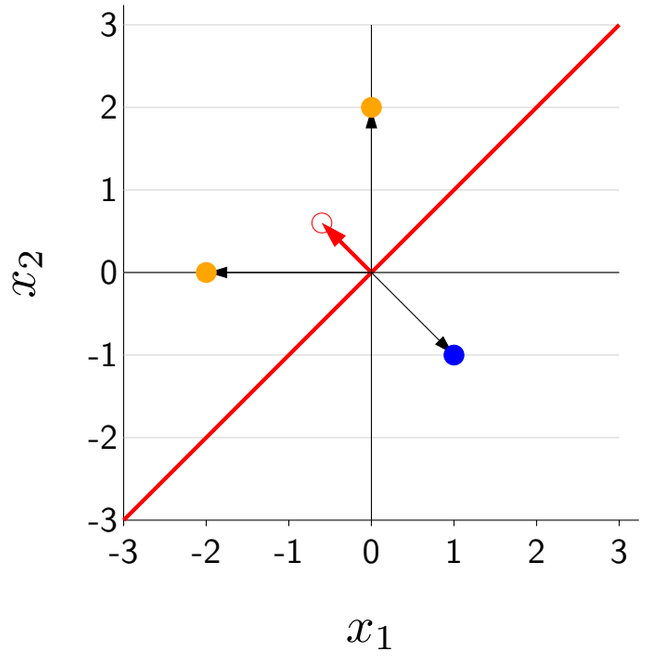
- We are going to go and set up our problem first..
- As usual, we are given **training data**, which consists of a set of examples. Each **example** consists of an input $x = (x_1, x_2)$ and a binary output y. We are considering two-dimensional inputs now to make the example a bit more interesting. The examples can be plotted, with the color denoting the label (orange for $+1$ and blue for -1).
- We still want a learning algorithm that takes the training data and produces a model f, that can predict whether y should be positive or negative. We will call this a **classifier**..
- We can visualize a classifier on the 2D plot by its **decision boundary**, which divides the input space into two regions: the region of input points that the classifier would output $+1$ and the region that the classifier would output -1. By convention, the arrow points to the positive region.
- Again, there are the same three design decisions to fully specify the learning algorithm:
- First, which classifiers f is the learning algorithm allowed to produce? Must the decision boundary be straight or can it curve? In other words, what is the **hypothesis class**?
- Second, how does the learning algorithm judge which classifier is good? In other words, what is the **loss function**?
- Finally, how does the learning algorithm actually find the best classifier? In other words, what is the **optimization algorithm**?

An example linear classifier

$$f(x) = \text{sign}(\overbrace{[-0.6, 0.6]}^{\mathbf{w}} \cdot \overbrace{[x_1, x_2]}^{\phi(x)})$$

$$\text{sign}(z) = \begin{cases} +1 & \text{if } z > 0 \\ -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \end{cases}$$

x_1	x_2	$f(x)$
0	2	1
-2	0	1
1	-1	-1



$$f([0, 2]) = \text{sign}([-0.6, 0.6] \cdot [0, 2]) = \text{sign}(1.2) = 1$$

$$f([-2, 0]) = \text{sign}([-0.6, 0.6] \cdot [-2, 0]) = \text{sign}(1.2) = 1$$

$$f([1, -1]) = \text{sign}([-0.6, 0.6] \cdot [1, -1]) = \text{sign}(-1.2) = -1$$

Decision boundary: x such that $\mathbf{w} \cdot \phi(x) = 0$

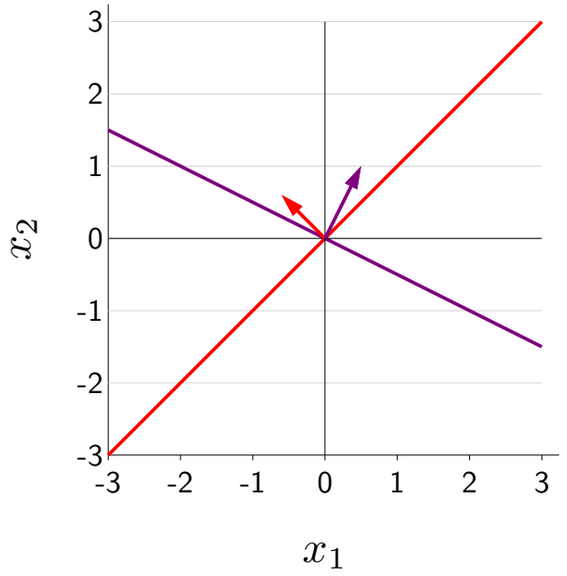
- Before we talk about the hypothesis class over all classifiers, we will start by exploring the properties of a specific linear classifier.
- First take the dot product between a fixed weight vector $\mathbf{w}$ and the input x, which we write as the identity feature map phi.
- The **sign** function of a number z is $+1$ if $z > 0$ and -1 if $z < 0$ and 0 if $z = 0$.
- Let's now visualize what f does. Notice on the plot the red arrow corresponds to the location of the weight vector w
- Let's feed some inputs into f.
- Take the first point (0, 2), which can be visualized on the plot. The vector from the origin to (0,2) and the vector to the weight vector (-0.6,0.6) forms an acute angle. This means that the dot product between them should be positive, and the classifier will return a positive label.
- The second point also forms an acute angle and therefore is classified as $+1$.
- The third point forms an obtuse angle and is classified as -1.
- Now you can hopefully see the pattern now. All points in the top left region (consisting of points forming an acute angle) will be classified $+1$, and all points in the bottom right (consisting of points forming an obtuse angle) will be classified -1.
- Points x which form a right angle with the weight vector has a zero dot product and is the **decision boundary**.
- Indeed, you can see pictorially that the decision boundary is perpendicular to the weight vector.

Hypothesis class: which classifiers?

$$\phi(x) = [x_1, x_2]$$

$$f(x) = \text{sign}([-0.6, 0.6] \cdot \phi(x))$$

$$f(x) = \text{sign}([0.5, 1] \cdot \phi(x))$$



General binary classifier:

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

Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^2\}$$

- We've looked at one particular red classifier.
- We can also consider an alternative purple classifer, which has a different decision boundary.
- In general for binary classification, given a particular weight vector $\mathbf{w}$ we define $f_{\mathbf{w}}$ to be the sign of the dot product.

Loss function: how good is a classifier?

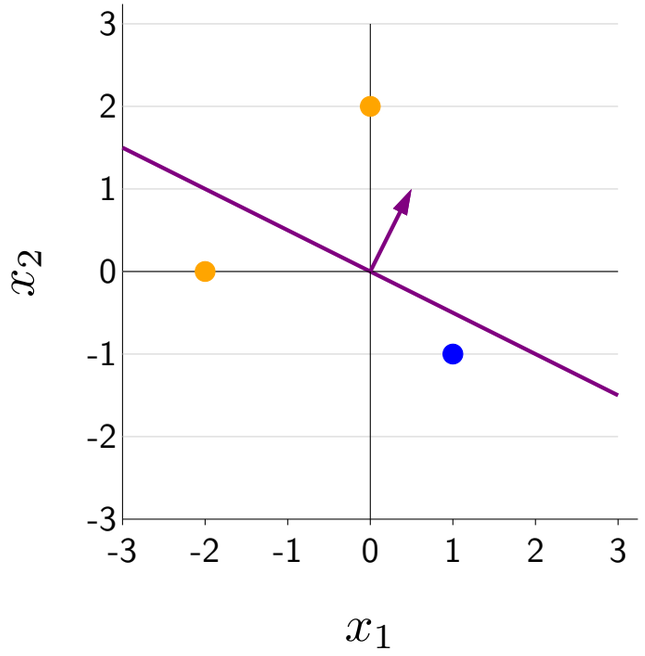
$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

$\mathbf{w} = [0.5, 1]$

$\phi(x) = [x_1, x_2]$

training data $\mathcal{D}_{\text{train}}$

x_1	x_2	y
0	2	1
-2	0	1
1	-1	-1



$$\text{Loss}_{0\text{-}1}(x, y, \mathbf{w}) = \mathbf{1}[f_{\mathbf{w}}(x) \neq y]$$ **zero-one loss**

$$\text{Loss}([0, 2], 1, [0.5, 1]) = \mathbf{1}[\text{sign}([0.5, 1] \cdot [0, 2]) \neq 1] = 0$$

$$\text{Loss}([-2, 0], 1, [0.5, 1]) = \mathbf{1}[\text{sign}([0.5, 1] \cdot [-2, 0]) \neq 1] = 1$$

$$\text{Loss}([1, -1], -1, [0.5, 1]) = \mathbf{1}[\text{sign}([0.5, 1] \cdot [1, -1]) \neq -1] = 0$$

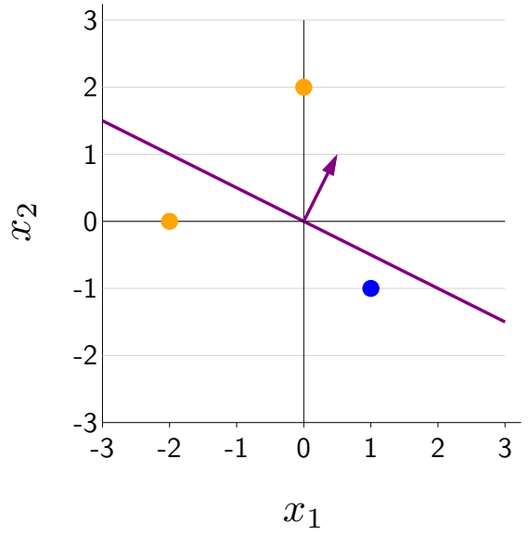
$$\text{TrainLoss}([0.5, 1]) = 0.33$$

- Now we proceed to the second design decision: the loss function, which measures how good a classifier is.
- Let us take the purple classifier, which can be visualized on the graph, as well as the training examples.
- Is this a good classifier? intuitively, it's not very good since it makes a mistake on the bottom left point To make this intuition precise, we will define the **zero-one loss** as a test for whether the classifier $f_{\mathbf{w}}(x)$ disagrees with the target label y. If so, then the indicator function - which we write at this bold numeral 1 with a bracket will return 1; otherwise, it will return 0.
- Let's see this classifier in action.
- For the first training example, the prediction is 1, the target label is 1, so the loss is 0.
- For the second training example, the prediction is -1, the target label is 1, so the loss is 1.
- For the third training example, the prediction is -1, the target label is -1, so the loss is 0.
- The total loss is simply the average over all the training examples, which yields $1/3$.

Score and margin

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

Target label: y



Definition: score

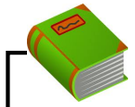
The score on an example (x, y) is $\mathbf{w} \cdot \phi(x)$, how **confident** we are in predicting $+1$.

Definition: margin

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

- Before we move to the third design decision (optimization algorithm), let us spend some time understanding two concepts so that we can rewrite the zero-one loss.
- Recall the definition of the predicted label and the target label.
- The first concept, which we already have encountered is the **score**. In regression, this is the predicted output, but in classification, this is the number before taking the sign.
- Intuitively, the score measures how confident the classifier is in predicting $+1$. If the weight vector is unit length, then this score is exactly the predicted sign times the distance of the input to the decision boundary.
- The second concept is **margin**, which measures how correct the prediction is. The larger the margin the more correct, and non-positive margins correspond to classification errors. If $y = 1$, then the score needs to be very positive for a large margin. If $y = -1$, then the score needs to be very negative for a large margin.
- The score and margin are fundamental building blocks for making loss functions The zero-one loss we have discussed is one type of loss you can build on top of the margin by reducing the margin to a zero-one value. We will see the value of thinking about the margin in the next few slides

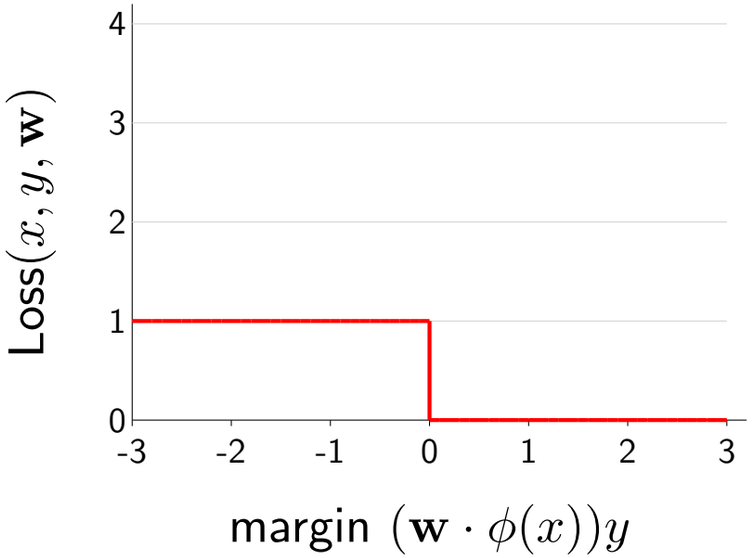
Zero-one loss rewritten



Definition: zero-one loss

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

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



- Now let us rewrite the zero-one loss in terms of the margin.
- We can also plot the loss against the margin.
- Again, a positive margin yields zero loss while a non-positive margin yields loss 1.

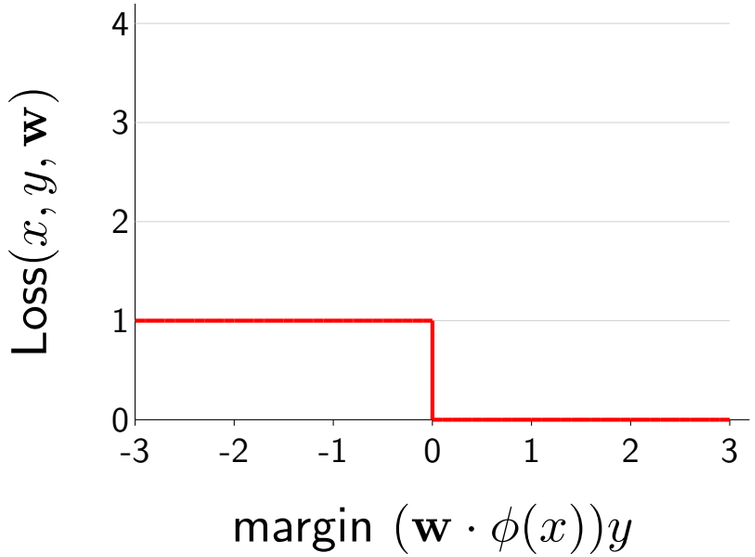
Optimization algorithm: how to compute best?

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

To run gradient descent, compute the gradient:

$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \nabla \text{Loss}_{\text{0-1}}(x, y, \mathbf{w})$$

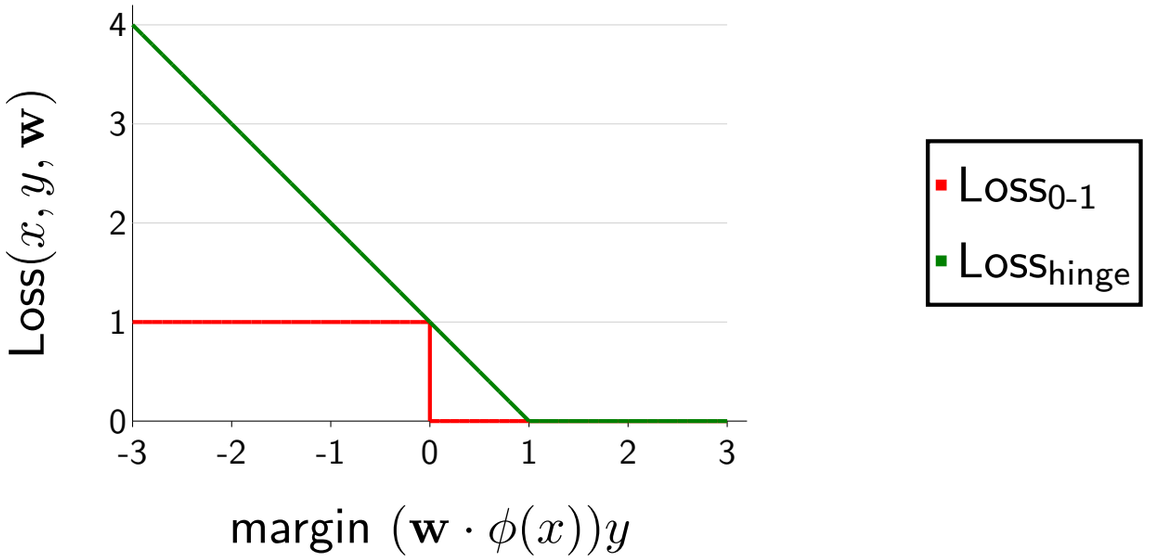
$$\nabla_{\mathbf{w}} \text{Loss}_{\text{0-1}}(x, y, \mathbf{w}) = \nabla \mathbf{1}[(\mathbf{w} \cdot \phi(x))y \leq 0]$$



Gradient is zero almost everywhere!

- Now we consider the third design decision, the optimization algorithm for minimizing the training loss.
- You might think this is going to be easy, and we can just do gradient descent This turns out to be surprisingly wrong Recall that to run gradient descent, we need to first compute the gradient.
- Just like before, we can take the gradient of the overall loss by summing the gradients of each example
- But this is where we run into problems: recall that the zero-one loss is flat almost everywhere (except at margin $= 0$), so the gradient is zero almost everywhere.
- What's the intuition? In the regression case, if we moved the weights a little bit, the losses for each point changed some points got closer, and some points got further. We could use this to find small tweaks to improve the weight. The zero-one loss is **different**: if change your weights a tiny bit and jiggle the decision boundary, your overall loss will not change unless you make a big enough change to make an example cross the decision boundary
- Because of this, you will be stuck if you try to use gradient descent on the zero one loss

Hinge loss



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

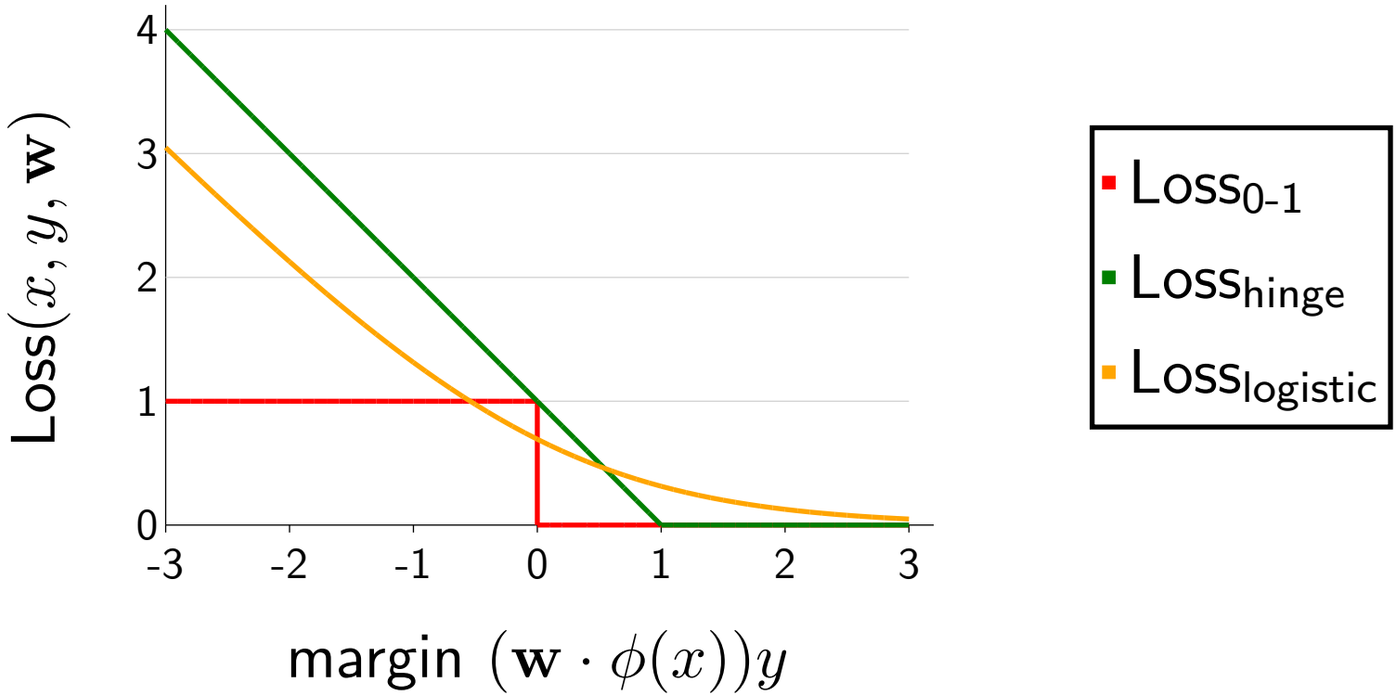
A point has zero loss if it is confidently classified correctly (margin ¿ 1)

Confidently misclassifying a point incurs a linear penalty

- To fix this problem, we have to choose another loss function.
- What was the problem with the zero one loss? it was that small changes to the weight didnt affect the loss at all. Ideally, we would be able to reduce the loss by making misclassified points closer to the decision boundary
- This idea is captured by the **hinge loss**, which is the maximum over a descending line and the zero function. It is best explained visually.
- The loss seems confusing at first, but its actually intuitive once you get to grips with it. There are two principles at work: In order to get zero loss, the model doenst just have to classify points correctly, it has to do so confidently This is why the loss is zero only if the margin is at least 1 On the other hand, if a model is too confident, and makes confident errors, it incurs a linearly increasing penalty
- From a theory perspective, the important property of the hinge is that it is an upper bound on the zero-one loss notice how the hinge line is always above the zero-one line in the plot This means that a small hinge loss guarantees a small zero-one loss. In particular, if the hinge loss is zero, the zero-one loss must also be zero.

Digression: logistic regression

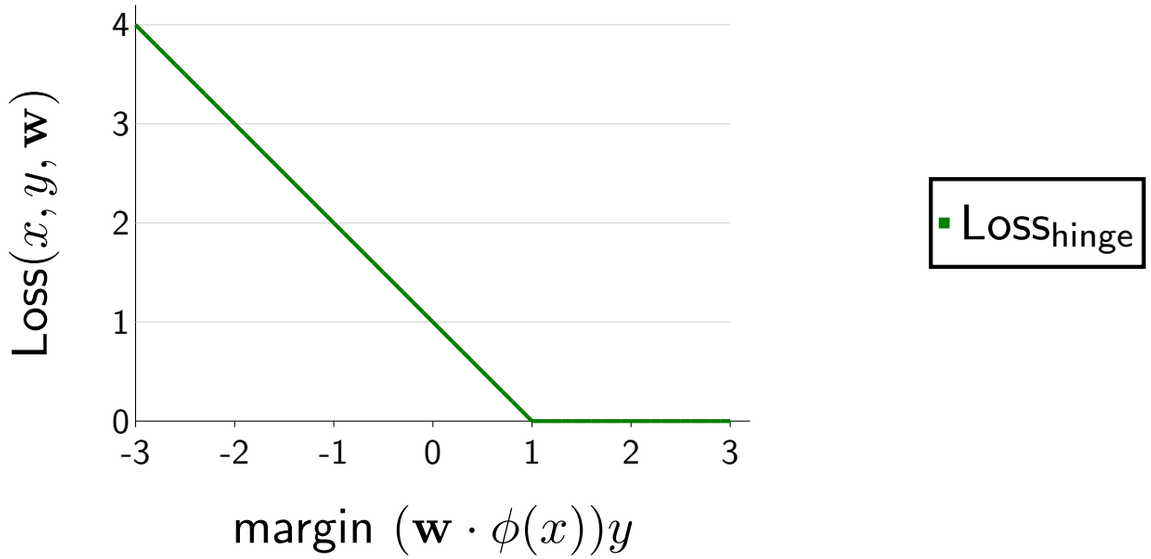
$$\text{Loss}_{\text{logistic}}(x, y, \mathbf{w}) = \log(1 + e^{-(\mathbf{w} \cdot \phi(x))y})$$



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

- Another popular loss function used in machine learning, especially in neural networks, 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 loss down by increasing the margin.

Gradient of the hinge loss



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

$$\nabla \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \begin{cases} -\phi(x)y & \text{if } 1 > \{(\mathbf{w} \cdot \phi(x))y\} \\ 0 & \text{otherwise} \end{cases}$$

- Now let us come back to optimization We said before that the zero-one loss was intractable. Is the same true of the hinge?
- Lets try to write down the gradient. This is a piecewise function, so lets reason about each piece separately Starting with the left side, when the margin is less than 1 then the hinge is equal to one minus the margin so the gradient with respect to w will be the negative of the feature vector times y
- On the right side when the margin is larger than 1, we have a flat zero function so the gradient is also zero
- Combining the two cases we end up with a piecewise function that is linear when the margin is less than one and zero otherwise Notice that unlike in the zero-one loss case, the gradients can only be zero if our predictor gets all of the points confidently correct
- What about when the margin is exactly 1? Technically, the gradient doesn't exist because the hinge loss is not differentiable there. But it turns out you can take either $-\phi(x)y$ or 0 and gradient descent will still work, thanks to the theory of what is known as subgradients.

Hinge loss on training data

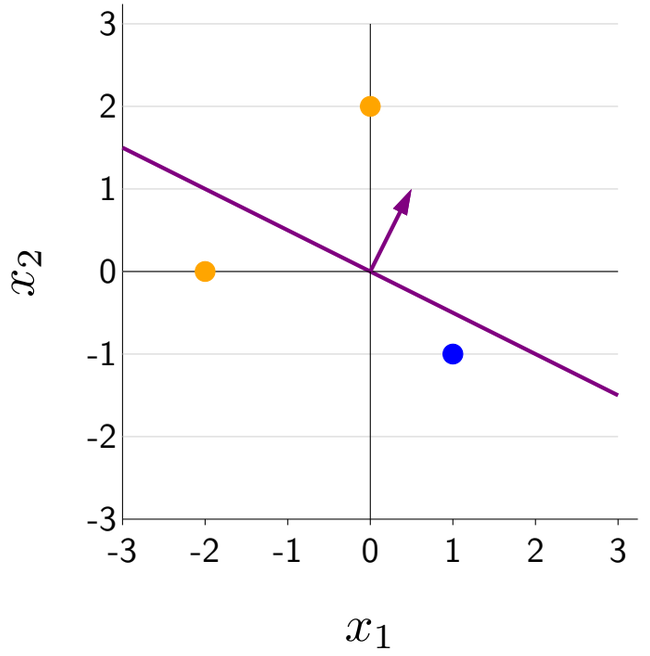
$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

$\mathbf{w} = [0.5, 1]$

$\phi(x) = [x_1, x_2]$

training data $\mathcal{D}_{\text{train}}$

x_1	x_2	y
0	2	1
-2	0	1
1	-1	-1



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

$\text{Loss}([0, 2], 1, [0.5, 1]) = \max\{1 - [0.5, 1] \cdot [0, 2](1), 0\} = 0$

$\text{Loss}([-2, 0], 1, [0.5, 1]) = \max\{1 - [0.5, 1] \cdot [-2, 0](1), 0\} = 2$

$\text{Loss}([1, -1], -1, [0.5, 1]) = \max\{1 - [0.5, 1] \cdot [1, -1](-1), 0\} = 0.5$

$\text{TrainLoss}([0.5, 1]) = 0.83$

$\nabla\text{Loss}([0, 2], 1, [0.5, 1]) = [0, 0]$

$\nabla\text{Loss}([-2, 0], 1, [0.5, 1]) = [2, 0]$

$\nabla\text{Loss}([1, -1], -1, [0.5, 1]) = [1, -1]$

$\nabla\text{TrainLoss}([0.5, 1]) = [1, -0.33]$

- Now let us revisit our earlier setting with the hinge loss.
- For each example (x, y), we can compute its loss, and the final loss is the average.
- For the first example of (0,2), we get this point correct with a marign of 2, so the loss is zero, and therefore the gradient is zero.
- For the second example of (-2,0), the loss is non-zero which is expected since the classifier is incorrect.
- For the third example of (1,-1), note that the loss is non-zero even though the classifier is correct. This is because the margin in this case is only 0.5 which is less than the required margin of 1.

Summary so far

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

	Regression	Classification
Prediction $f_{\mathbf{w}}(x)$	score	sign(score)
Relate to target y	residual (score $- y$)	margin (score y)
Loss functions	squared absolute deviation	zero-one hinge logistic
Algorithm	gradient descent	gradient descent

- Let us end by comparing and contrasting linear classification and linear regression.
- The score is a common quantity that drives the prediction in both cases.
- In regression, the output is the raw score. In classification, the output is the sign of the score.
- To assess whether the prediction is correct, we must relate the score to the target y. In regression, we use the residual, which is the difference (lower is better). In classification, we use the margin, which is the product (higher is better).
- Given these two quantities, we can form a number of different loss functions. In regression, we studied the squared loss, but we could also consider the absolute deviation loss (taking absolute values instead of squared). In classification, we care about the zero-one loss (which corresponds to the missclassification rate), but we optimize the hinge or the logistic loss.
- Finally, gradient descent can be used in both settings.

Roadmap

Topics in the lecture:

Linear regression

Linear classification

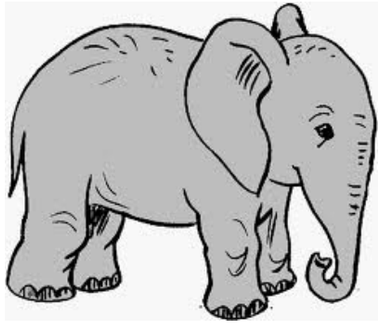
Stochastic gradient descent

Losses over groups (group DRO)

- In this module, we will introduce stochastic gradient descent, which has completely taken over optimization for machine learning in the last decade.

Gradient descent for large datasets

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



Algorithm: gradient descent

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

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

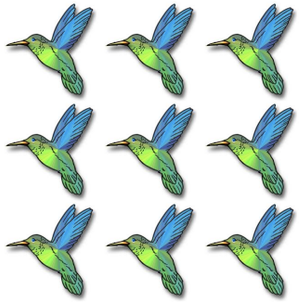
$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

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

- So far, we've seen gradient descent as a general-purpose algorithm to optimize the training loss.
- There's one issue when applying gradient descent to modern large-scale datasets All of our training losses are averages over the dataset
- So each step of gradient descent is going to require going through the entire dataset Since you most likely want few hundred to a thousand gradient descent steps, this can get very expensive fast
- Can we get algorithms that do not require looking through the entire dataset to make updates?

Stochastic gradient descent

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



Algorithm: stochastic gradient descent

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

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

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

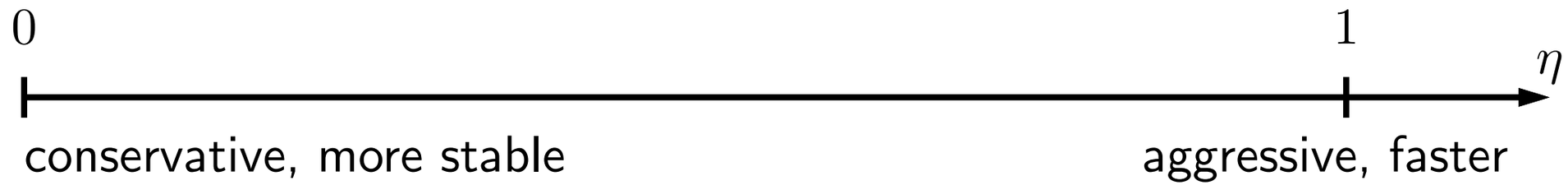
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$

- The answer is **stochastic gradient descent** often referred to as (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.
- Aside: there is a continuum between SGD and GD called minibatch SGD, where each update consists of an average over a random sample of B examples.
- Aside: There are many many variants of SGD, with choices of how and whether to shuffle or sample your data or incorporating ideas like momentum. There are too many variants to cover here, but Id be happy to cover them separately if any of you are curious

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?



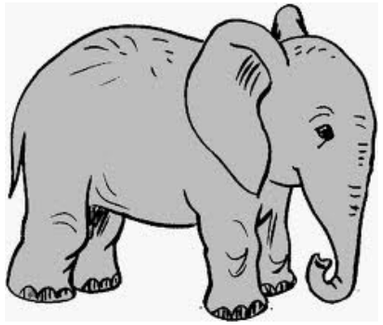
Strategies:

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

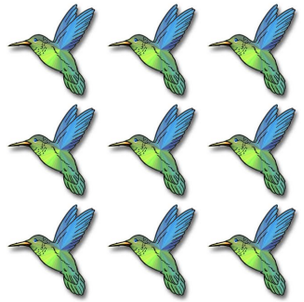
- One remaining issue is choosing the step size, which in practice is quite important.
- Generally, larger step sizes are like driving fast. You can get faster convergence, but you might also get very unstable results and crash and burn.
- On the other hand, with smaller step sizes you get more stability, but you might get to your destination more slowly. Note that the weights do not change if $\eta = 0$
- A somewhat popular 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. This lets the optimizer initially make big updates, but eventually leads to smaller fine-tuning updates at the end
- Aside: There are more sophisticated algorithms like AdaGrad and Adam that adapt the step size based on the data, so that you don't have to tweak it as much.

Summary

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



gradient descent



stochastic gradient descent

Key idea: stochastic updates

Trade off between update **quality**, and **quantity**.

- In summary, we've shown how stochastic gradient descent can be faster than gradient descent.
- Gradient just spends too much time refining its gradient (quality), while you can get a quick and dirty estimate just from one sample and make more updates (quantity).
- The optimal choice for a problem is often somewhere in the middle, where techniques such as mini-batching, are used to find a sweet spot where the gradient updates are low-noise but still cheap to compute.

Roadmap

Topics in the lecture:

Linear regression

Linear classification

Stochastic gradient descent

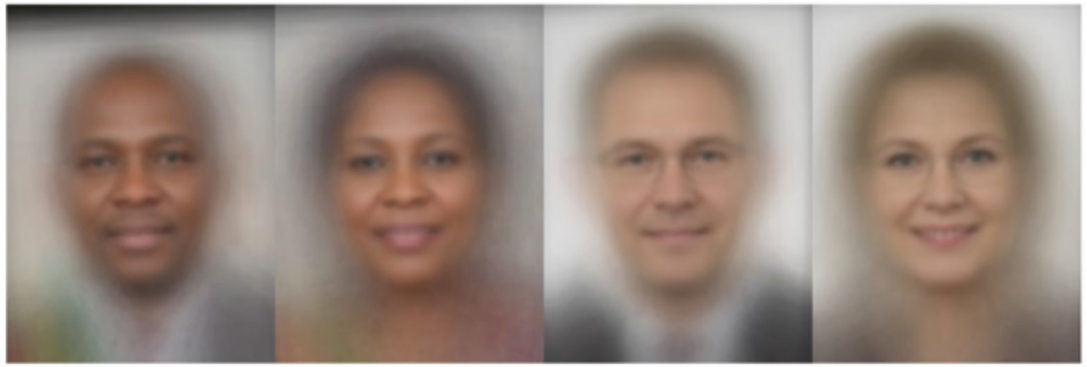
Losses over groups (group DRO)

- Thus far, we have focused on finding predictors that minimize the training loss, which is an average (of the loss) over the training examples.
- While averaging seems reasonable, in this module, I'll show that averaging can be problematic and lead to inequalities in accuracy across groups.
- Then I'll briefly present an approach called **group distributional robust optimization (group DRO)**, which can mitigate some of these inequalities.

[Buolamwini & Gebru 2018]

Gender Shades

Gender Classifier	Darker Male	Darker Female	Lighter Male	Lighter Female	Largest Gap
Microsoft	94.0%	79.2%	100%	98.3%	20.8%
FACE++	99.3%	65.5%	99.2%	94.0%	33.8%
IBM	88.0%	65.3%	99.7%	92.9%	34.4%



Inequalities arise in machine learning

- is the Gender Shades project by Joy Buolamwini and Timnit Gebru (paper).
- In this project, they collected an evaluation dataset of face images, which aimed to have balanced representation of both faces with lighter and darker skin tones, and across men and women. To do this they curated the public face images from members of parliament across several African and European countries.
- Then, they evaluated three commercial systems, from Microsoft, Face++, and IBM, on the task of gender classification (which in itself maybe a dubious task, but was one of the services offered by these companies).
- The results were striking: all three systems got nearly 100% accuracy for lighter-skinned males, but they all got much worse accuracy for darker-skinned females.
- Gender Shades is just one of many examples pointing at a general and systemic problem: machine learning models, which are typically optimized to maximize average accuracy, can yield poor accuracies for certain **groups** (subpopulations).
- These groups can be defined by protected classes as given by discrimination law such as race and gender, or perhaps defined by the user identity or location.

[from a New York Times article]

False arrest due to facial recognition

Real-life consequences

- Poor accuracy of machine learning systems can have serious real-life consequences. In one vivid case, a Black man by the name of Robert Julian-Borchak Williams was wrongly arrested due to a incorrect match with another Black man captured from a surveillance video, and this mistake was made by a facial recognition system.
- Given the Gender Shades project, we can see that lower accuracies for some groups might even lead to more arrests, which adds to the already problematic inequalities that exist in our society today.
- In this module, we'll focus only on performance disparities in machine learning and how we can mitigate them.
- But even if facial recognition worked equally well for all groups of people, should it even be used for law enforcement? Should it be used at all? These are bigger ethical questions, and it's always important to remember that sometimes, the issue is not with the solution, but the framing of the problem itself.

Linear regression with groups

training data

	x	y	g
example	1	4	A
example	2	8	A
example	5	5	B
	6	6	B
	7	7	B
	8	8	B

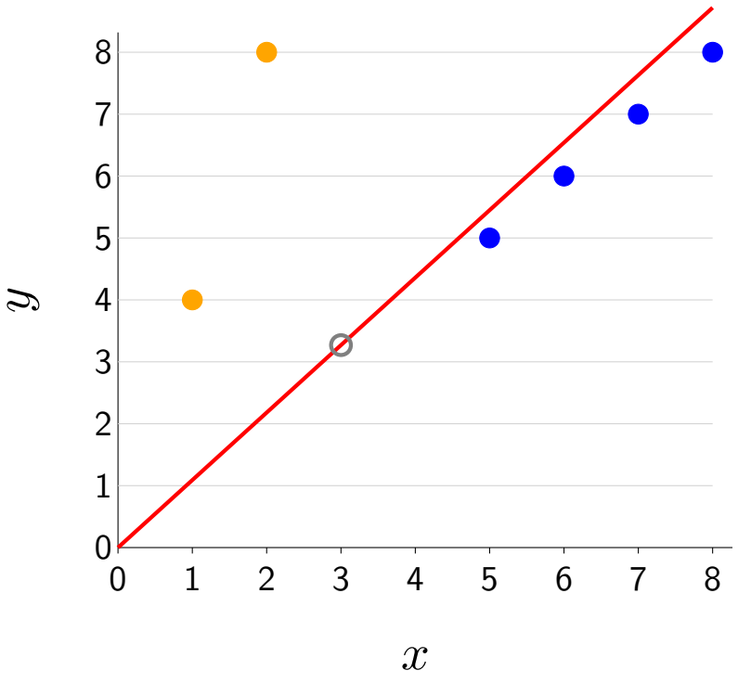
group

learning algorithm

3 input

f predictor

3.27 output



$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) \qquad \mathbf{w} = [w] \qquad \phi(x) = [x]$$

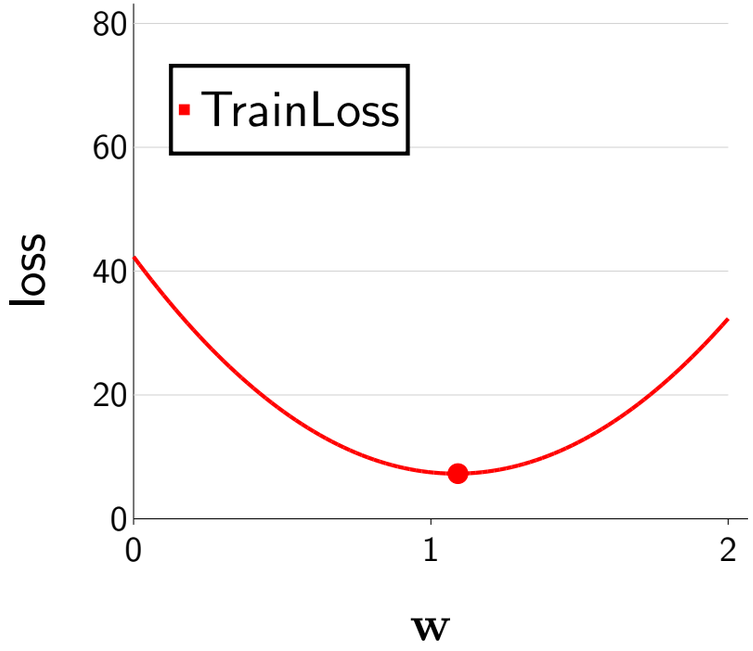
Note: predictor $f_{\mathbf{w}}$ does not use group information g

- Gender Shades was an example of classification, but to make things simpler, let us consider linear regression.
- We start with training data, but this time, each example consists of not only the input x and the output y, but also a **group** g (e.g., gender).
- In this example, we will assume we have two groups, A and B. As we see on the plot, the A points rise quickly, and the B points lie on the identity line, so the points behave differently.
- Recall the goal of regression is to produce a predictor that can take in a new input and predict an output.
- In linear regression, each predictor $f_{\mathbf{w}}$ computes the dot product of the weight vector $\mathbf{w}$ and a feature vector $\phi(x)$, and for this example, we define the feature map $\phi(x)$ to be the identity map, so that our hypothesis class consists of lines that pass through the origin.
- Looking ahead a bit, we see that there is a bit of a tension, where what slope w is best for group A is not the same as what is best for group B. The question is how we can compromise.
- Note that in this setting, the predictor $f_{\mathbf{w}}$ does not use the group information g, so it cannot explicitly specialize to the different groups. The group information is only used by the learning algorithm as well as to evaluate the performance across different groups.

Average loss

$$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$$

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B



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

$$\text{TrainLoss}(1) = \tfrac{1}{6}((1-4)^2 + (2-8)^2 + (5-5)^2 + (6-6)^2 + (7-7)^2 + (8-8)^2) = 7.5$$

- Recall that in regression, we typically use the squared loss, which measures how far away the prediction $f_{\mathbf{w}}(x)$ is away from the target y.
- Also recall that we defined the training loss to be an average of the per-example losses. This gives us a loss value for each value of $\mathbf{w}$ (see plot).
- So if we evaluate the training loss at $\mathbf{w} = 1$, we're averaging over the 6 examples. For each example, the prediction $f_{\mathbf{w}}(x)$ is just x (since $\mathbf{w} \cdot \phi(x) = [1] \cdot [x] = x$, so we can average the squared differences between x and y, producing a final answer of 7.5.
- If you evaluate $\mathbf{w}$ at a different value, you get a different training loss.
- If we minimize the training loss, then we can find this point $\mathbf{w} = 1.09$.

Per-group loss

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B

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

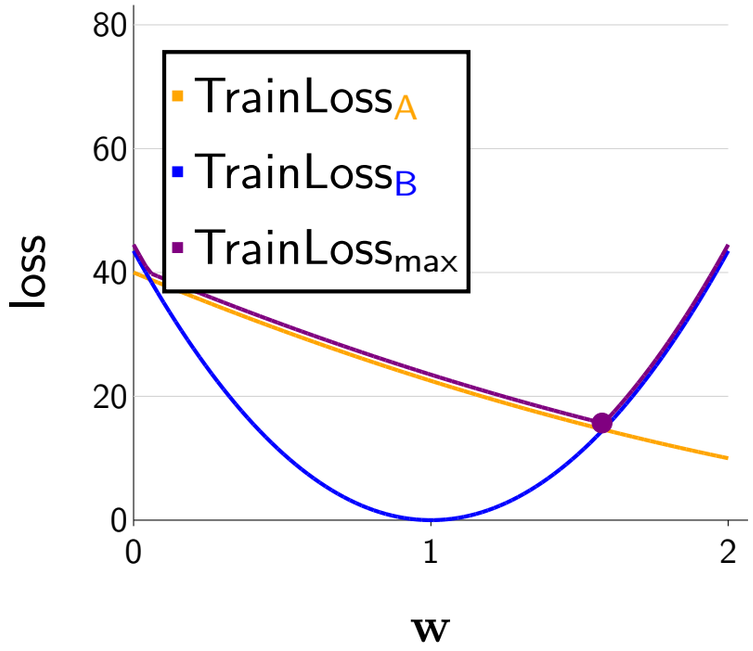
$$\text{TrainLoss}_A(1) = \tfrac{1}{2}((1-4)^2 + (2-8)^2) = 22.5$$

$$\text{TrainLoss}_B(1) = \tfrac{1}{4}((5-5)^2 + (6-6)^2 + (7-7)^2 + (8-8)^2) = 0$$

- Let us now take a careful look at the loss of each group.
- First, define $\mathcal{D}_{\text{train}}(g)$ to be the set of examples in group g. For example, $\mathcal{D}_{\text{train}}(\text{A}) = \{(1, 4), (2, 8)\}$.
- Then define the **per-group loss** TrainLoss_g of a weight vector $\mathbf{w}$ to be the average loss over the points in group g.
- If we choose $\mathbf{w} = [1]$ as our running example (because it's easy to compute), we see that the per-group loss on group A is 22.5, while the per-group loss on group B is 0.
- So note that even though the average loss was only 7.5, there is a huge performance disparity, with group A suffering a much larger loss.

Maximum group loss

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B



$$\text{TrainLoss}_{\text{max}}(\mathbf{w}) = \max_g \text{TrainLoss}_g(\mathbf{w})$$

$\text{TrainLoss}_A(1) = 22.5$

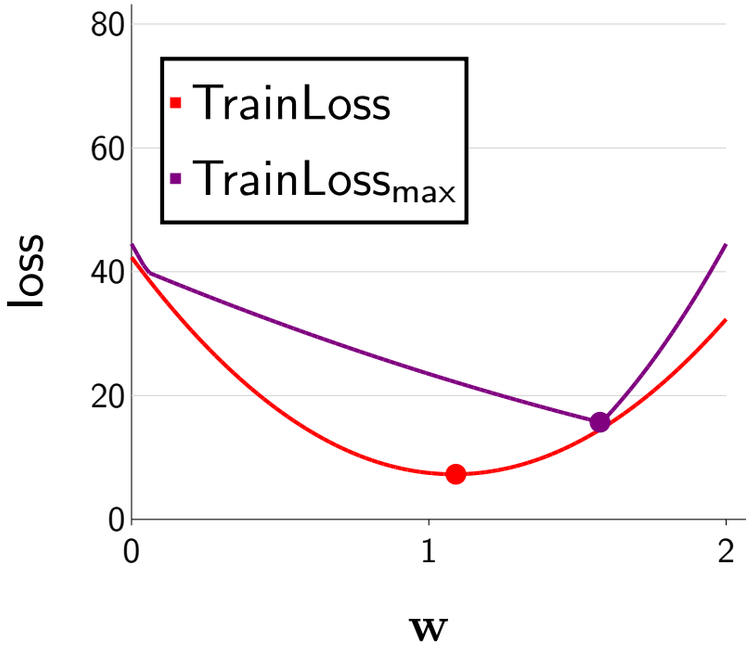
$\text{TrainLoss}_B(1) = 0$

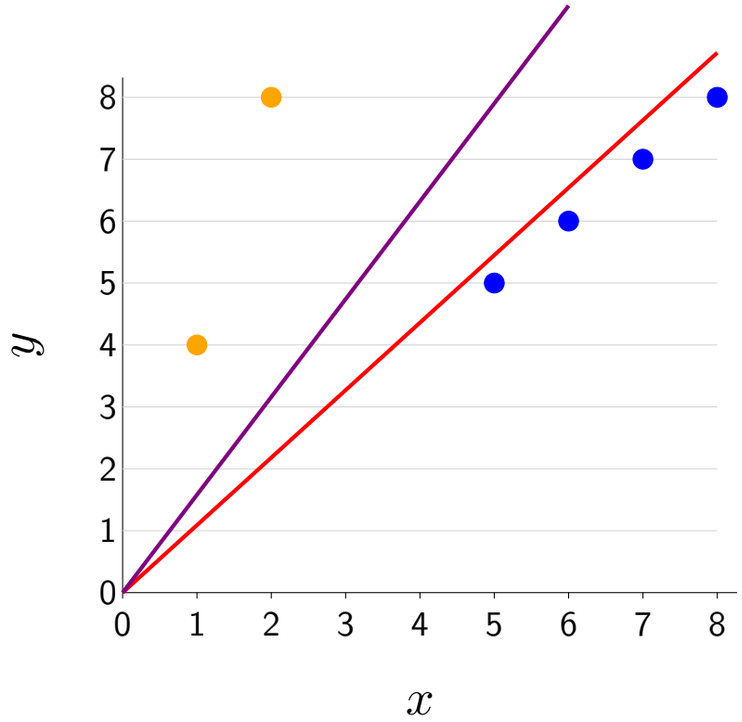
$\text{TrainLoss}_{\text{max}}(1) = \max(22.5, 0) = 22.5$

- We now want to somehow capture the per-group losses by one number. To do this, we look at the worst-case over groups.
- We now define the **maximum group loss** to be the largest over all groups. This function is the pointwise maximum of the per-group losses, which you can see on the plot as taking the upper envelope of the two per-group losses.
- We call this method group distributionally robust optimization (group DRO), because it is a special case of a broader framework Distributionally robust optimization (DRO).
- Going back to our running example with $\mathbf{w} = [1]$, we take the maximum of 22.5 and 0, which is 22.5.
- Note that this is much higher than the average loss (7.5), signaling that some group(s) are far less well off than the average.

Average loss versus maximum group loss

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B





Standard learning:

minimizer of average loss: $\mathbf{w} = 1.09$

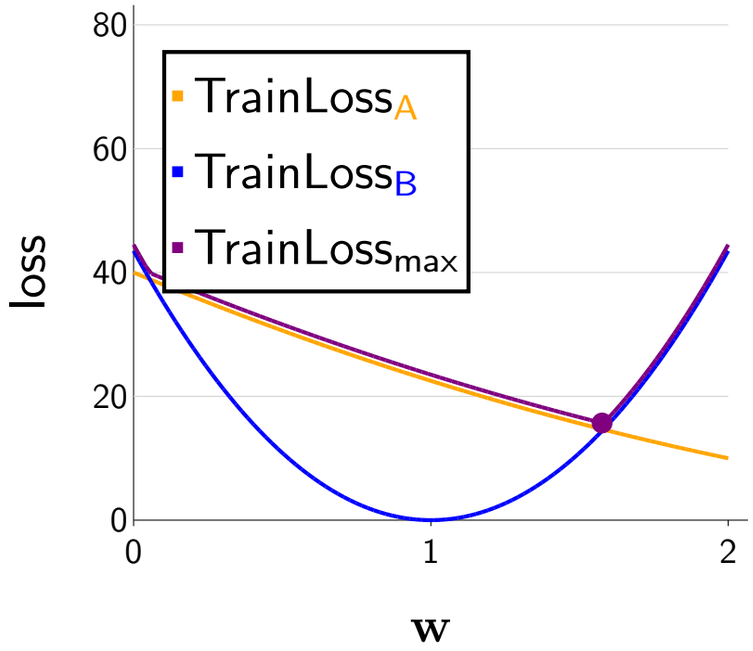
Group distributionally robust optimization (group DRO):

minimizer of maximum group loss: $\mathbf{w} = 1.58$

- Let us now compare the old average loss (the standard training loss) TrainLoss and the new maximum group loss $\text{TrainLoss}_{\text{max}}$.
- We minimize the average loss by setting the slope $\mathbf{w} = [1.09]$, yielding an average loss of 7.29. However, this setting gets much higher maximum group loss (over 20). Pictorially, we see that this $\mathbf{w}$ heavily favors the majority group (B).
- If instead we minimize the maximum group loss directly, we would choose $\mathbf{w} = 1.58$, yielding a maximum group loss of 15.59. Pictorially, we see that this solution pays more attention to (is closer to) the A points.
- Intuitively, the average loss favors majority groups over minority groups, but the maximum group loss gives a stronger voice to the minority groups, and as we see here, their influence is felt to a greater extent.

Training via gradient descent

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B



$$\text{TrainLoss}_{\text{max}}(\mathbf{w}) = \max_g \text{TrainLoss}_g(\mathbf{w})$$

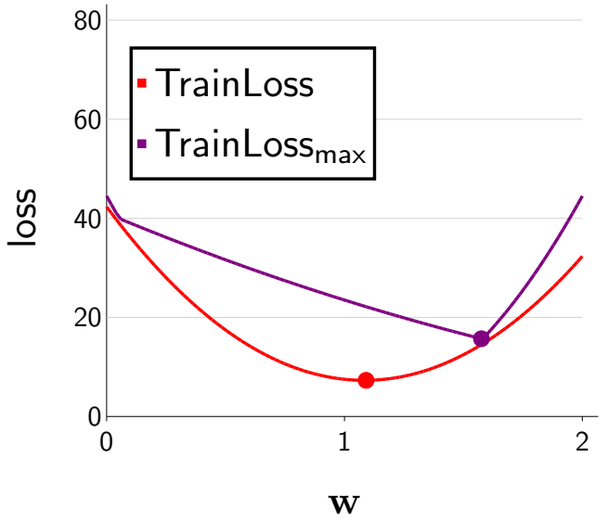
$$\nabla \text{TrainLoss}_{\text{max}}(\mathbf{w}) = \nabla \text{TrainLoss}_{g^*}(\mathbf{w})$$

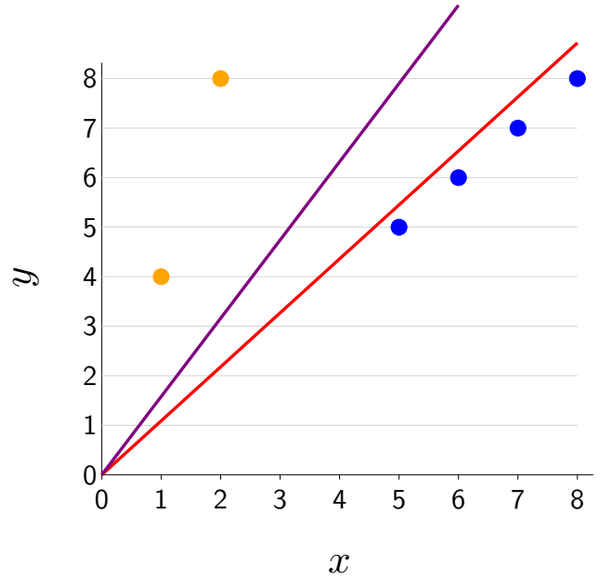
$$\text{where } g^* = \arg\max_g \text{TrainLoss}_g(\mathbf{w})$$

- In general, we can find minimize the maximum group loss by gradient descent.
- We just have to be able to take the gradient of $\text{TrainLoss}_{\text{max}}$, which is a maximum over the per-group losses.
- The gradient of a max is simply the gradient of the term that achieves the max.
- So algorithmically, it's very intuitive: you first compute whichever group (g^*) has the highest loss, and then you just evaluate the gradient only on per-group loss of that group (g^*).
- but you can see this paper for more details.

Summary

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B





- Maximum group loss $\neq$ average loss
- Group DRO: minimize the maximum group loss
- Many more nuances: intersectionality? don't know groups? overfitting?

- To summarize, we've introduced the setting where examples are associated with groups. We see that by default, doing well on average is not the same as doing well on all groups (in other words, average loss is not the same as the maximum group loss), and the optimal solution for one objective is not optimal for the other objective.
- We presented an approach, group DRO that can ensure that the worst-off group is doing well.
- One parting remark is that while group DRO offers a mathematically clean solution, there are many subtleties when applying this to the real world. Intersectionality refers to the fact that groups which are defined by the conjunction of multiple attributes (e.g., White woman) may behave differently than the groups defined by individual attributes. What if you don't even have group information? How do you deal with overfitting (we've only looked at the training loss)?
- These are questions that are beyond the scope of this module, but I hope this short module has raised the need to think about about inequality as a first-class citizen, and piqued your interest to learn more.
- For further reading, consider checking out the book Fairness and machine learning: Limitations and Opportunities,

Important topics and recap

Linear regression

Linear function class

Squared loss

Stochastic gradient descent

Quality-quantity tradeoff

Step size selection

Linear classification

Linear classifiers

Zero-one loss

Margin and score

Hinge and logistic losses

Worst-case losses over groups

Group DRO and worst-case losses

Optimizing the worst-case loss

- This may have been a lot to take in, so lets take a moment to go over the important big picture ideas
- We covered some general machine learning ideas, including the 3-part decomposition of a problem into a hypothesis class, loss function, and optimizer. The hypothesis is the set of allowed predictors. The loss measures how good these predictors are, and the optimizer is a way of finding the best predictor We also introduced gradient descent, a general-purpose approach for optimizing ML models
- In the linear regression part, we learned what a linear hypothesis looks like, how to express it as a dot product. We wrote down the squared loss, which is a useful way of measuring the quality of a regression fit and computed its gradient
- In the classification part, we learned how to express a linear decision boundary. As well as define and compute the zero-one loss, which has some clear problems. We defined the margin and score, which are useful ways of defining alternative losses Finally, we defined hinge and logistic loss using the margin and score
- In the SGD part, we asked the question - how can we train these models on big datasets and discussed how iteration complexity as a function of data size is important we went through SGD - a way of using just one example to do updates and presented this as a way of getting faster, but lower quality updates