

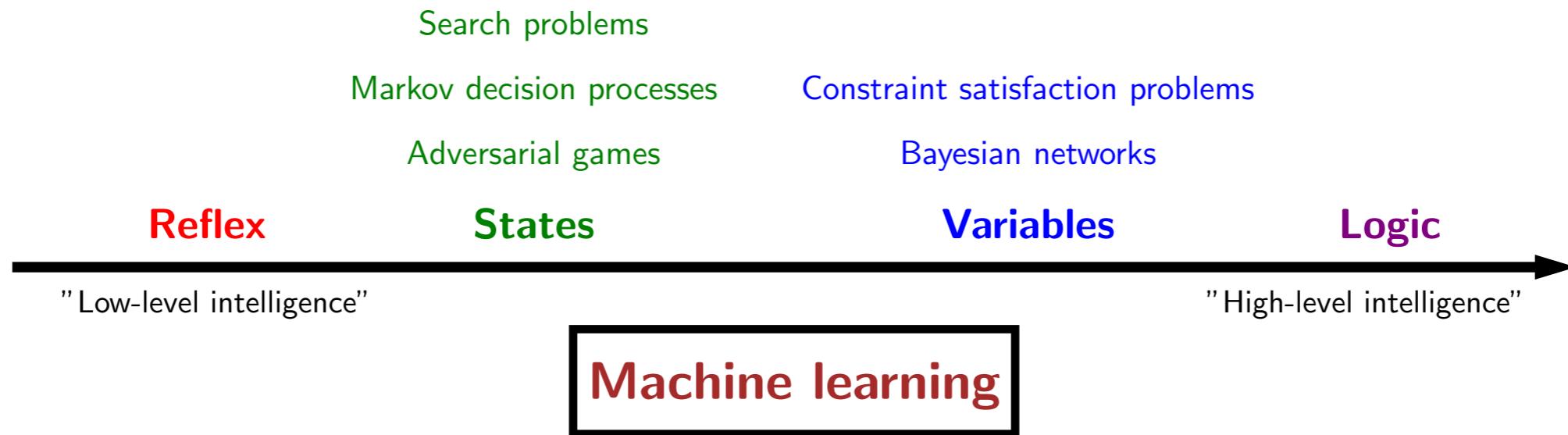


Machine learning: overview



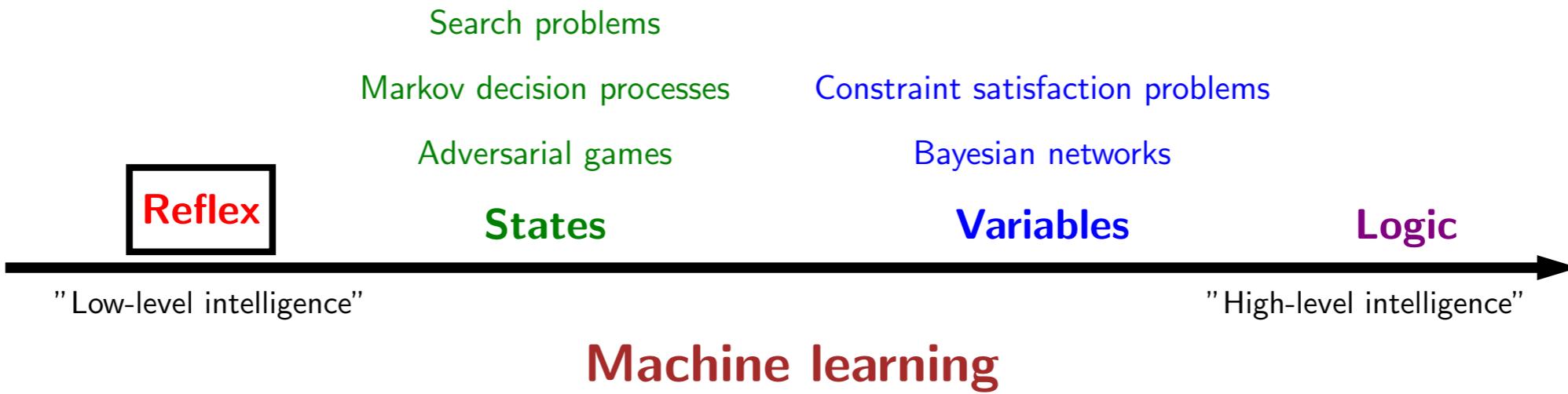
- In this module, I will provide an overview of the topics we plan to cover under machine learning.

Course plan



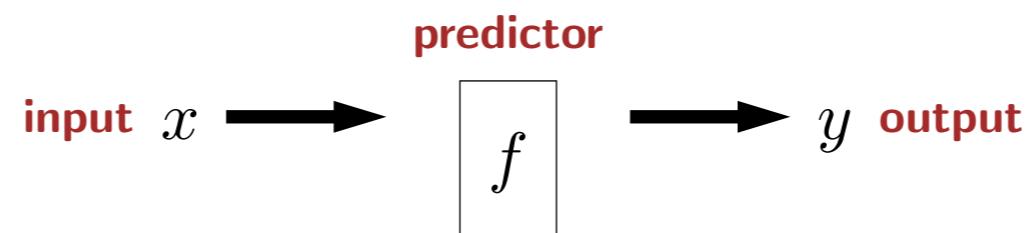
- Recall that machine learning is the process of turning data into a model. Then with that model, you can perform inference on it to make predictions.

Course plan



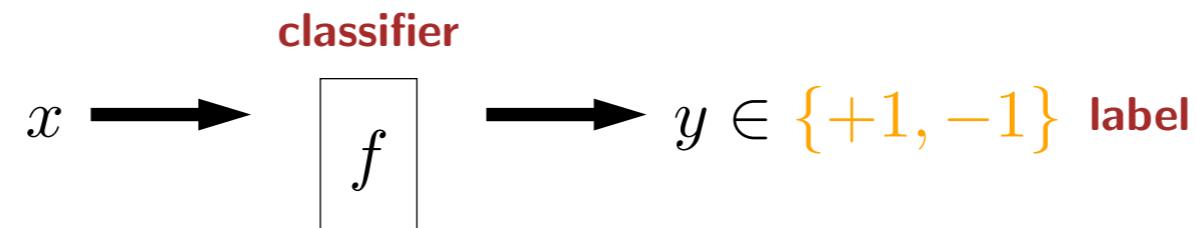
- While machine learning can be applied to any type of model, we will focus our attention on reflex-based models, which include models such as linear classifiers and neural networks.
- In reflex-based models, inference (prediction) involves a fixed set of fast, feedforward operations.

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 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**.

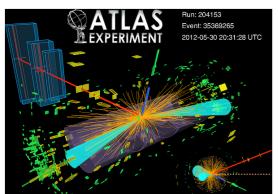
Binary classification



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



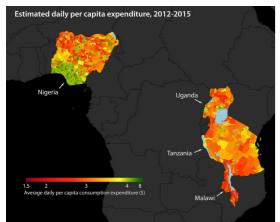
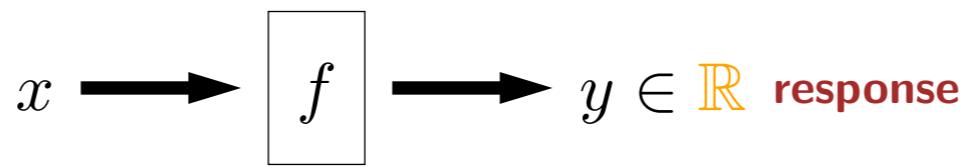
Toxic comments: online comment → toxic or not toxic



Higgs boson: measurements of event → 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 fraudulent 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



Poverty mapping: satellite image \rightarrow asset wealth index



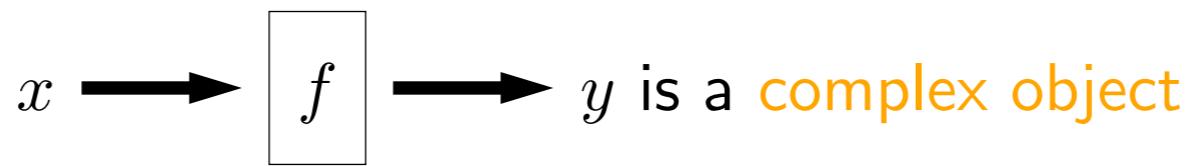
Housing: information about house \rightarrow price



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



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!
- Aside: one challenge with this approach is that the errors might cascade: if you start making errors, then you might go off the rails and start making even more errors.

Roadmap

Models

Tasks

Linear regression

Linear classification

K-means

Non-linear features

Feature templates

Neural networks

Differentiable programming

Algorithms

Stochastic gradient descent

Backpropagation

Considerations

Generalization

Best practices

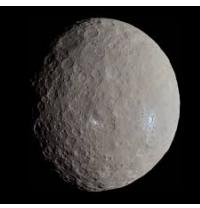
- Here are the rest of the modules under the machine learning unit.
- We will start by talking about regression and binary classification, the two most fundamental tasks in machine learning. Specifically, we study the simplest setting: **linear regression** and **linear classification**, where we have linear models trained by gradient descent.
- Next, we will introduce **stochastic gradient descent**, and show that it can be much faster than vanilla gradient descent.
- Then we will push the limits of linear models by showing 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 needed for training without having to take gradients manually.
- 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. We take a brief detour and discuss **K-means**, which is a simple unsupervised learning algorithm for clustering data points.
- We end on a more reflective note: **Generalization** is about answering the question: 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.



Machine learning: linear regression



- In this module, we will cover the basics of 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

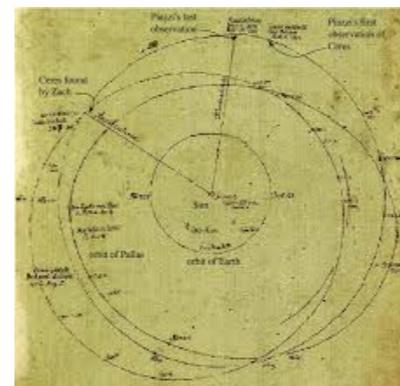
When and where will Ceres be observed again?

- 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

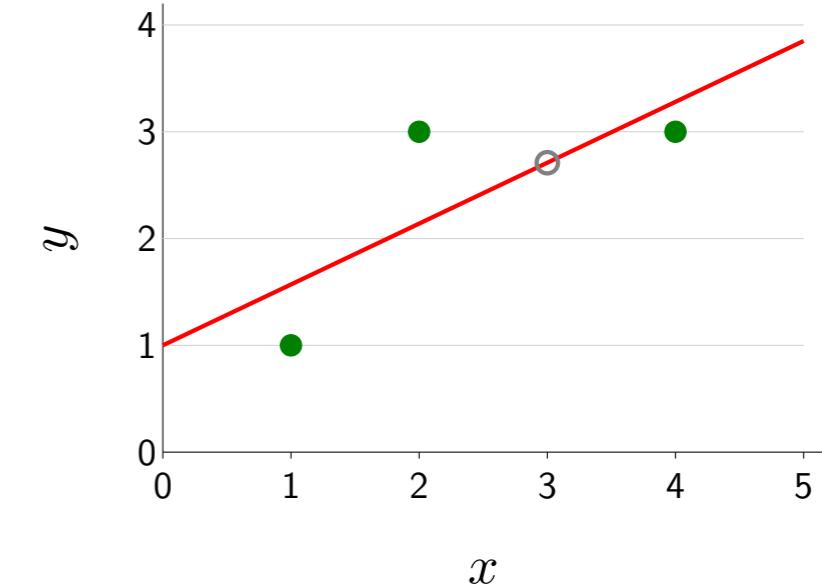
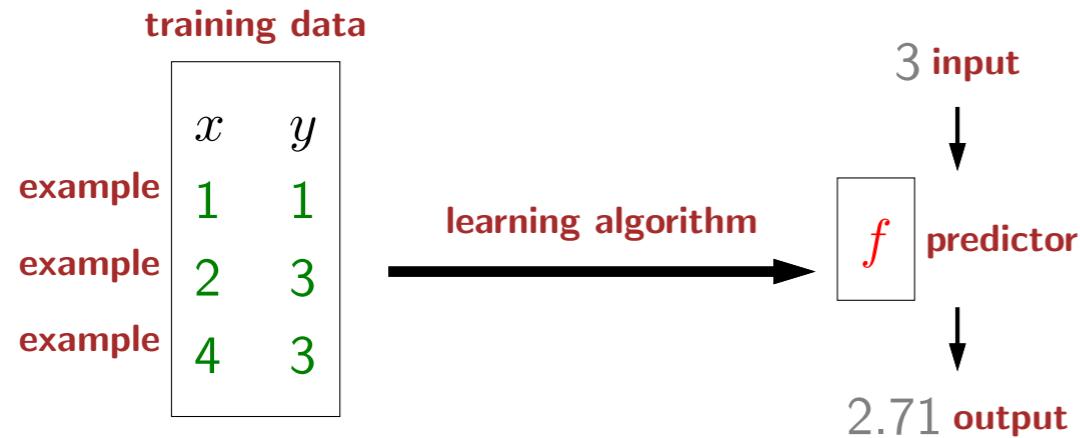


December 7, 1801: Ceres located within 1/2 degree of Gauss's prediction, much more accurate than other astronomers

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**

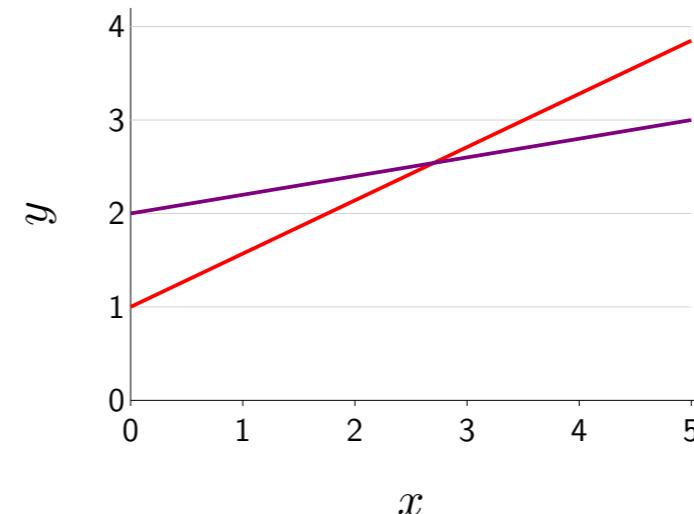
- Let us now present the linear regression framework.
- Suppose we are given **training data**, which consists of a set of examples. Each **example** (also known as data point, instance, case) consists of an input x and an output y . We can visualize the training set by plotting y against x .
- A learning 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? Only lines or curves as well? In other words, what is the **hypothesis class**?
- Second, how does the learning algorithm judge which predictor is good? In other words, what is the **loss function**?
- Finally, how does the learning algorithm actually find the best predictor? In other words, what is the **optimization algorithm**?

Hypothesis class: which predictors?

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

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

$$f(x) = w_1 + w_2x$$



Vector notation:

$$\text{weight vector } \mathbf{w} = [w_1, w_2]$$

$$\text{feature extractor } \phi(x) = [1, x] \text{ feature vector}$$

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

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

Hypothesis class:

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

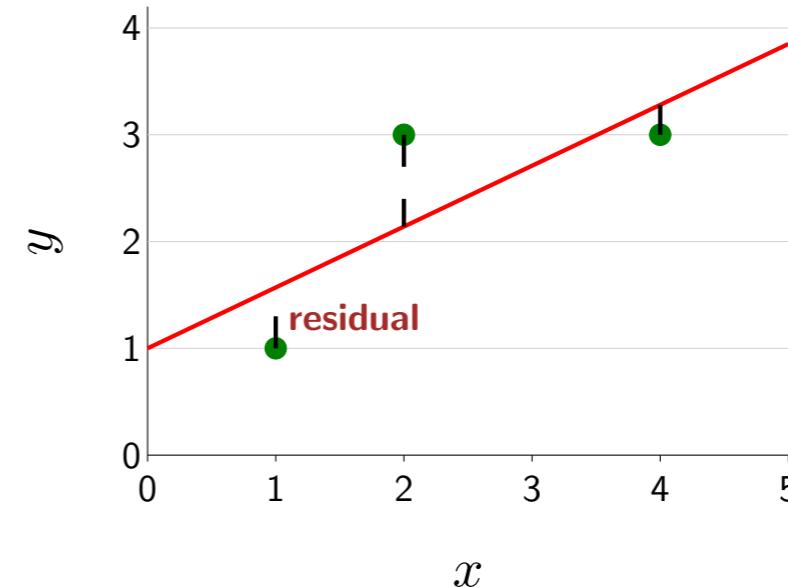
- Let's consider the first design decision: what is the hypothesis class? One possible predictor is the red line, where the intercept is 1 and the slope is 0.57, Another predictor is the purple line, where the intercept is 2 and the slope is 0.2.
- In general, let's consider all predictors of the form $f(x) = w_1 + w_2x$, where the intercept w_1 and the slope w_2 can be arbitrary real numbers.
- Now let us generalize this further using vector notation. Let's pack the intercept and slope into a single vector, which we will call the **weight vector** (more generally called 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** $[1, x]$.
- Now we can succinctly write the predictor f_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.
- Finally, define the hypothesis class \mathcal{F} to be simply the set of all possible predictors f_w as we range over all possible weight vectors w . This is the possible functions that we want our learning algorithm to consider.

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



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

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

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

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

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

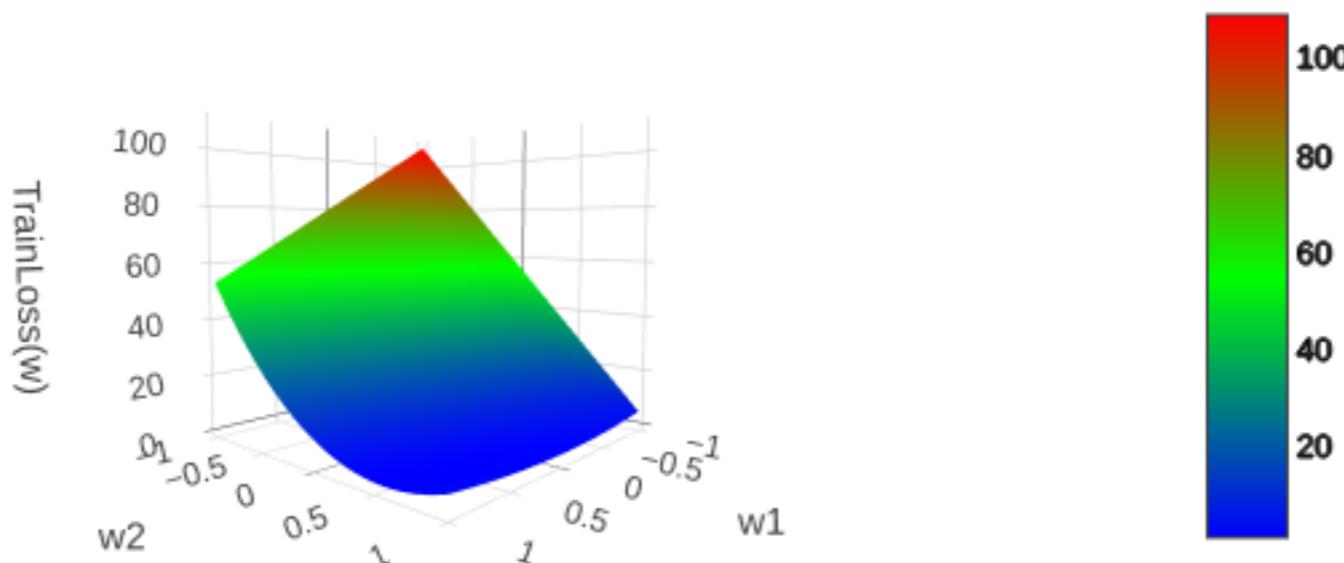
$$\text{TrainLoss}([1, 0.57]) = 0.38_{10}$$

- The next design decision is how to judge each of the many possible predictors.
- 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 $f_w(x)$ and the actual output y , known as the **residual**.
- Now define the **loss function** on given example with respect to 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.
- Aside: You might wonder why we are taking the square of the residual as opposed to taking an absolute value of the residual (known as the absolute deviation): the answer for now is both mathematical and computational convenience, though if your data potentially has outliers, it is beneficial to use the absolute deviation.
- For each example, we 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 wish to solve the optimization problem.



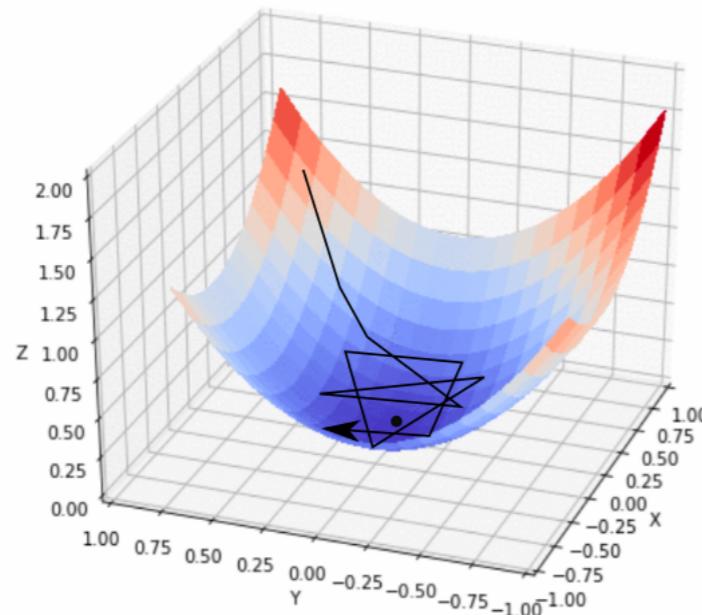
Optimization algorithm: how to compute best?

Goal: $\min_w \text{TrainLoss}(w)$



Definition: gradient

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



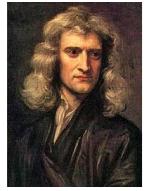
Algorithm: gradient descent

Initialize $w = [0, \dots, 0]$

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

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

- Now the third design decision: how do we compute the best predictor, i.e., the solution to the optimization problem?
- To answer this question, we can actually forget that we're doing linear regression or machine learning. We simply have an objective function $\text{TrainLoss}(\mathbf{w})$ that we wish to minimize.
- We will adopt the "follow your nose" strategy, i.e., **iterative optimization**. We start with some \mathbf{w} and keep on tweaking it to make the objective function go down.
- To do this, we will rely on the gradient of the function, which tells us the direction to move in that will decrease the objective function the most.
- Formally, this iterative optimization procedure is called **gradient descent**. We first initialize \mathbf{w} to some value (say, all zeros).
- Then perform the following update T times, where T is 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 and the number of epochs T are two **hyperparameters** of the optimization algorithm, which we will discuss later.



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 $\text{TrainLoss}(\mathbf{w})$.
- 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.
- The gradient of a sum is the sum of the gradient, the gradient of an expression squared is twice that expression times the gradient of that expression, and 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 $\phi(x)$.
- Aside: no matter what the loss function is, the gradient is always something times $\phi(x)$ because \mathbf{w} only affects the loss through $\mathbf{w} \cdot \phi(x)$.

Gradient descent example

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

x	y
1	1
2	3
4	3

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

Gradient update: $\mathbf{w} \leftarrow \mathbf{w} - 0.1 \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

t	$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$	\mathbf{w}
		[0, 0]
1	$\underbrace{\frac{1}{3}(2([0, 0] \cdot [1, 1] - 1)[1, 1] + 2([0, 0] \cdot [1, 2] - 3)[1, 2] + 2([0, 0] \cdot [1, 4] - 3)[1, 4])}_{=[-4.67, -12.67]}$	[0.47, 1.27]
2	$\underbrace{\frac{1}{3}(2([0.47, 1.27] \cdot [1, 1] - 1)[1, 1] + 2([0.47, 1.27] \cdot [1, 2] - 3)[1, 2] + 2([0.47, 1.27] \cdot [1, 4] - 3)[1, 4])}_{=[2.18, 7.24]}$	[0.25, 0.54]
...
200	$\underbrace{\frac{1}{3}(2([1, 0.57] \cdot [1, 1] - 1)[1, 1] + 2([1, 0.57] \cdot [1, 2] - 3)[1, 2] + 2([1, 0.57] \cdot [1, 4] - 3)[1, 4])}_{=[0, 0]}$	[1, 0.57]

- Let's step through an example of running gradient descent.
- Suppose we have the same dataset as before, the expression for the gradient that we just computed, and the gradient update rule, where we take the step size $\eta = 0.1$.
- We start with the weight vector $w = [0, 0]$. Let's then plug this into the expression for the gradient, which is an average over the three training examples, and each term is the residual (prediction - target) times the feature vector.
- That vector is multiplied by the step size (0.1 here) and subtracted out of the weight vector.
- We then take the new weight vector and plug it in again to the expression for the gradient. This produces another gradient, which is used to update the weight vector.
- If you run this procedure for long enough, you eventually get the final weight vector. Note that the gradient at the end is zero, which indicates that the algorithm has converged and running it longer will not change anything.

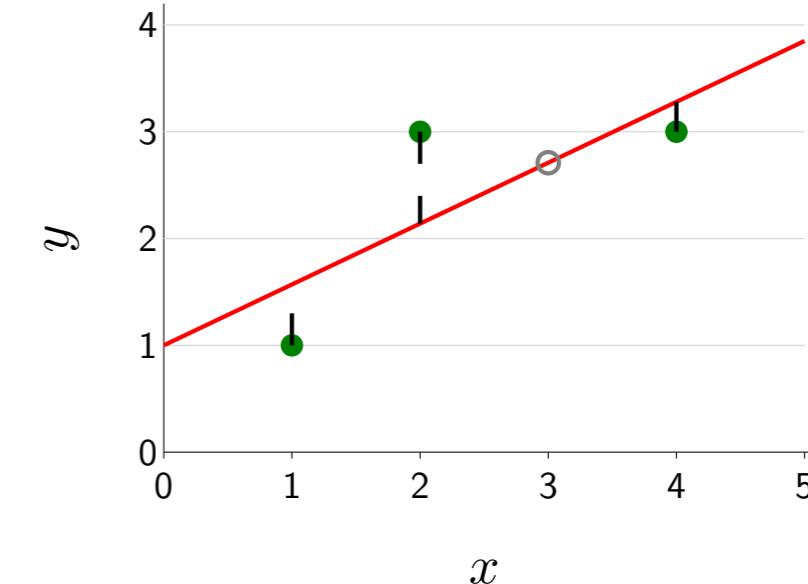
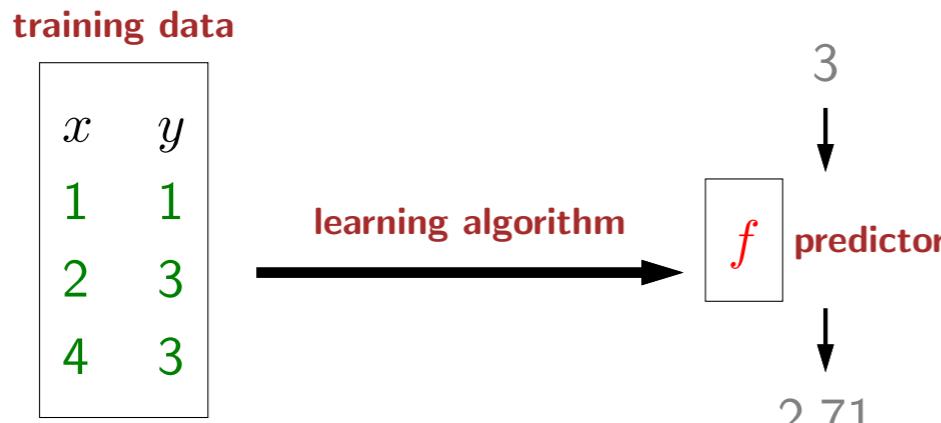
Gradient descent in Python

[code]

- To make things even more concrete, let's code up gradient descent in Python.
- In practice, you would probably use TensorFlow or PyTorch, but I will go with a very bare bones implementation on top of NumPy just to emphasize how simple the ideas are.
- Note that in the code, we are careful to separate out the **optimization problem** (what to compute), that of minimizing training loss on the given data, from the **optimization algorithm** (how to compute it), which works generically with the weight vector and gradients of the objective function.



Summary



Which predictors are possible?

Hypothesis class

How good is a predictor?

Loss function

How to compute best predictor?

Optimization algorithm

Linear functions

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

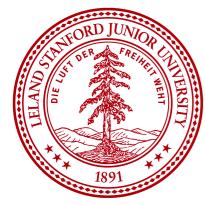
Squared loss

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

Gradient descent

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

- 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.

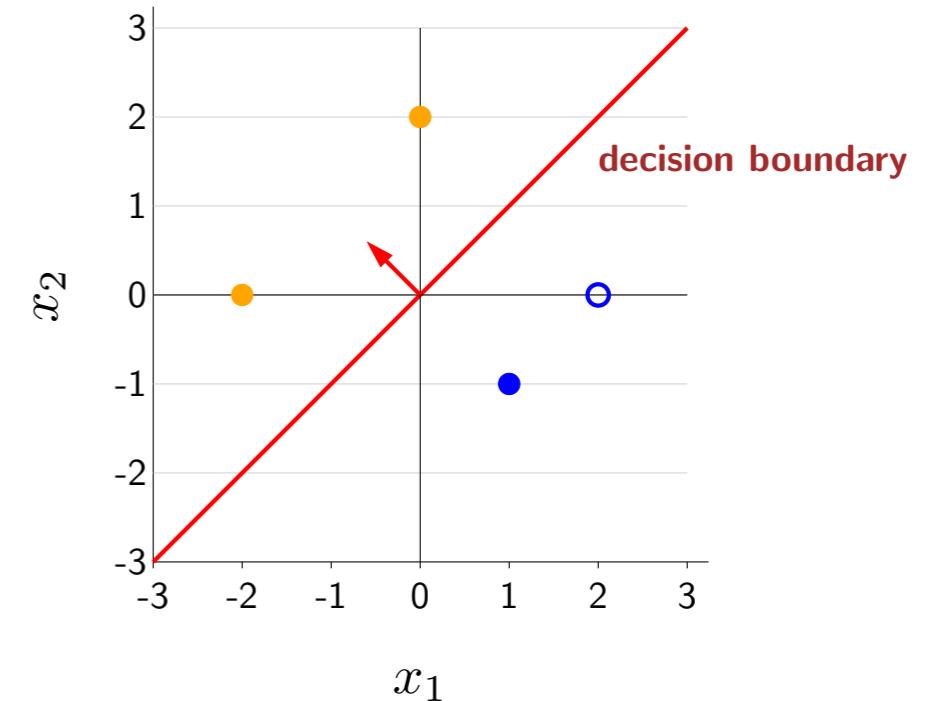
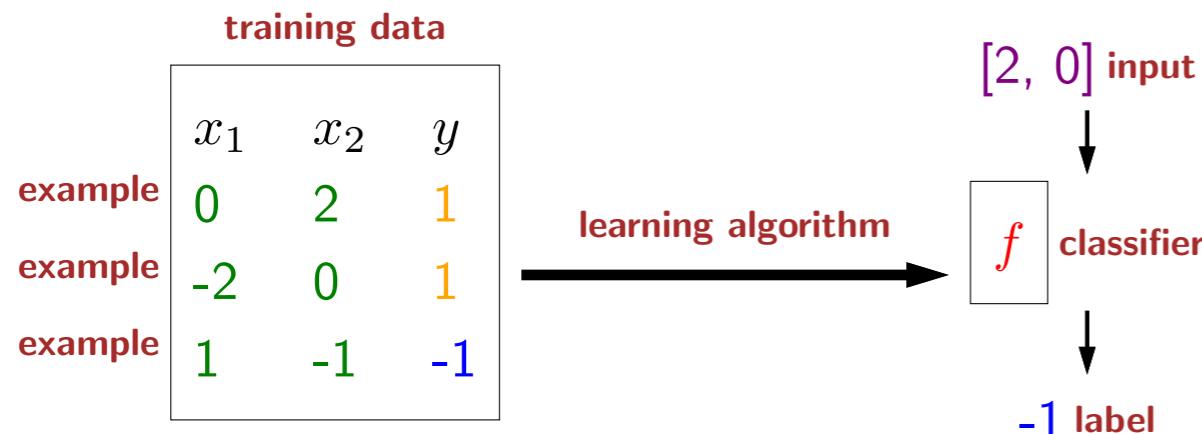


Machine learning: linear classification



- 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**

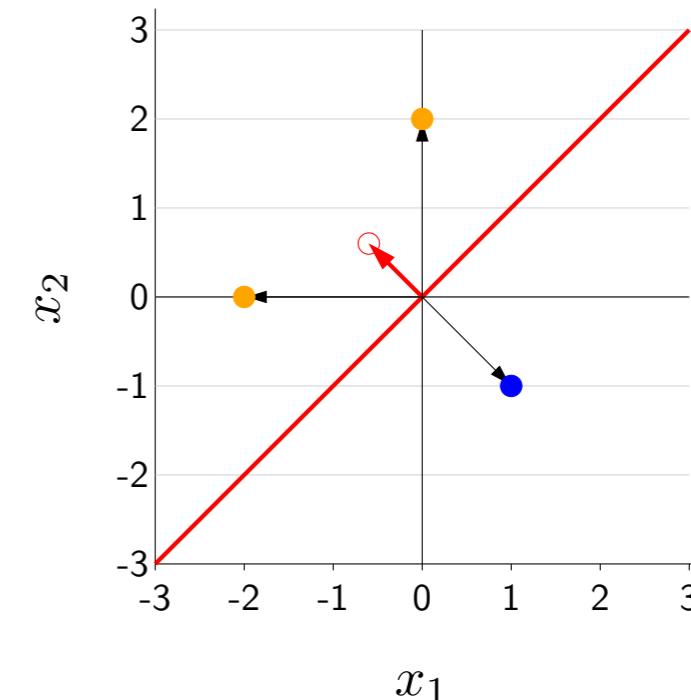
- Here is the linear classification framework.
- As usual, we are given **training data**, which consists of a set of examples. Each **example** consists of an input $x = (x_1, x_2)$ and an output y . We are considering two-dimensional inputs now to make the example a bit more interesting. The examples 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 , which we will call a **classifier** in the context of classification.
- The classifier takes a new input and produces an output. 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 $\mathbf{w} \cdot \phi(x) = 0$

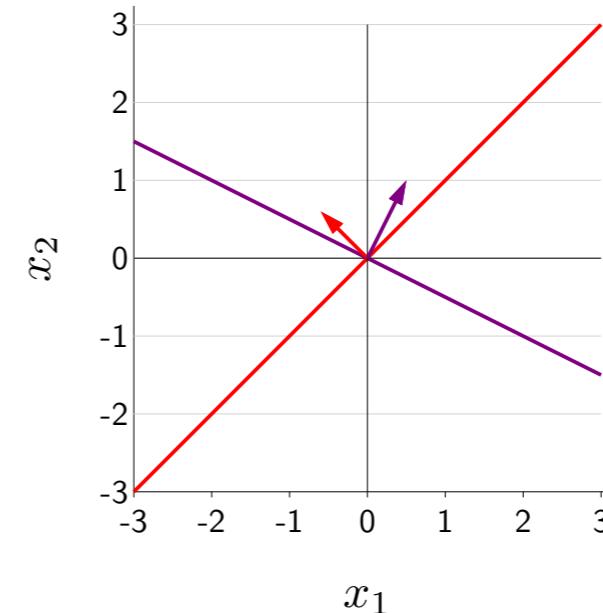
- Before we talk about the hypothesis class over all classifiers, we will start by exploring the properties of a specific linear classifier.
- First take the dot product between a fixed weight vector \mathbf{w} and the identity feature vector $\phi(x)$. Then take the sign of the dot product.
- The **sign** 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. First, we can plot \mathbf{w} either as a point or as a vector from the origin to that point; the latter will be most useful for our purposes.
- Let's feed some inputs into f .
- Take the first point, which can be visualized on the plot as a vector. Recall from linear algebra that the dot product between two vectors is the cosine of the angle between them. In particular, the dot product is positive iff the angle is acute and negative iff the angle is obtuse. The first point forms an acute angle and therefore is classified as $+1$, which can also be verified mathematically.
- 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 this region (consisting of points forming an acute angle) will be classified $+1$, and all points in this region (consisting of points forming an obtuse angle) will be classified -1 .
- Points x which form a right angle ($\mathbf{w} \cdot \phi(x) = 0$) form 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:

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

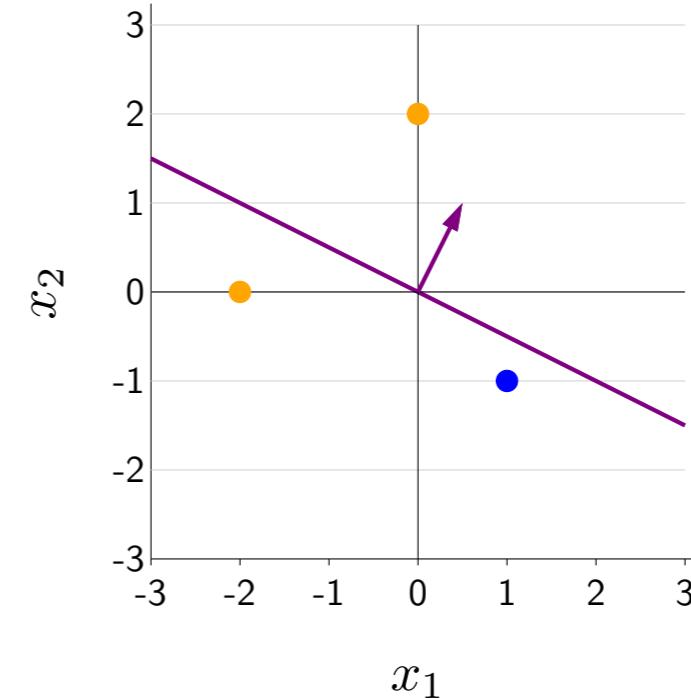
- We've looked at one particular red classifier.
- We can also consider an alternative purple classifier, which has a different decision boundary.
- In general for binary classification, given a particular weight vector w we define f_w to be the sign of the dot product.

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



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

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

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

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

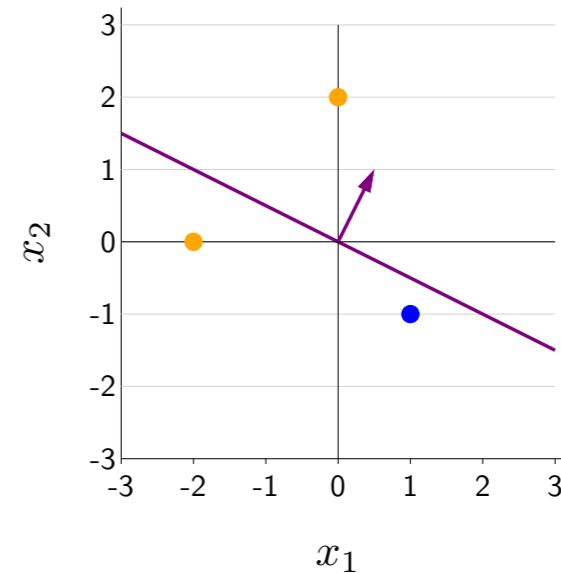
$$\text{TrainLoss}([0.5, 1]) = 0.33$$

- 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.
- Now we want to define a loss function that captures how the model predictions deviate from the data. We will define the **zero-one loss** to check if the model prediction $f_w(x)$ disagrees with the target label y . If so, then the indicator function $\mathbf{1}[f_w(x) \neq y]$ 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.

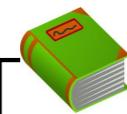


Definition: margin

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

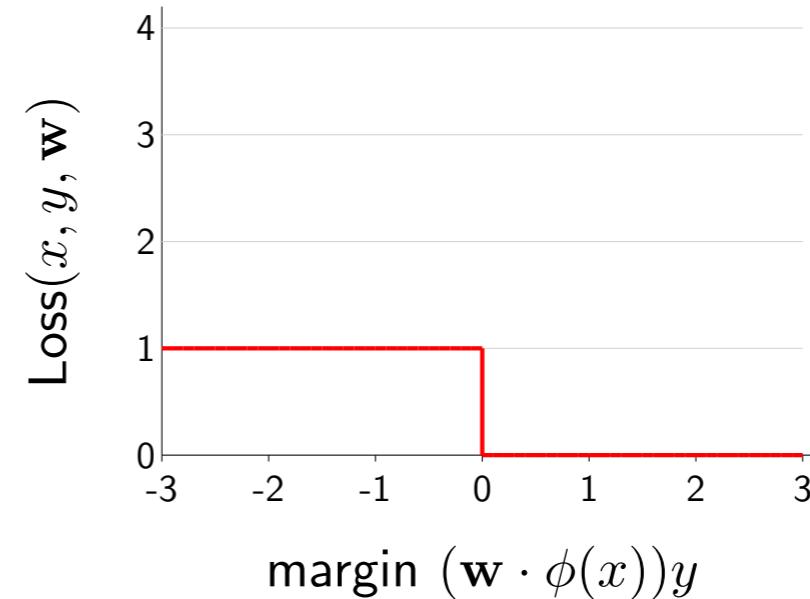
- Before we move to the third design decision (optimization algorithm), let us spend some time understanding two concepts so that we can rewrite the zero-one loss.
- Recall the definition of the predicted label and the target label.
- The first concept, which we already have encountered is the **score**. 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.
- Points farther away from the decision boundary have larger scores.
- 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.
- Note that if we look at the actual prediction $f_w(x)$, we can only ascertain whether the prediction was right or not.
- By looking at the score and the margin, we can get a more nuanced view into the behavior of the classifier.

Zero-one loss rewritten



Definition: zero-one loss

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



- Now let us 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.

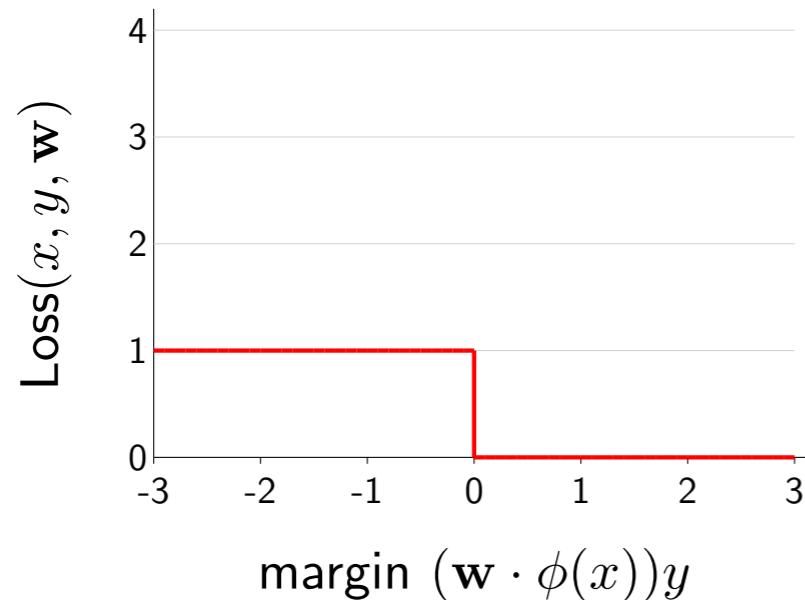
Optimization algorithm: how to compute best?

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

To run gradient descent, compute the gradient:

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

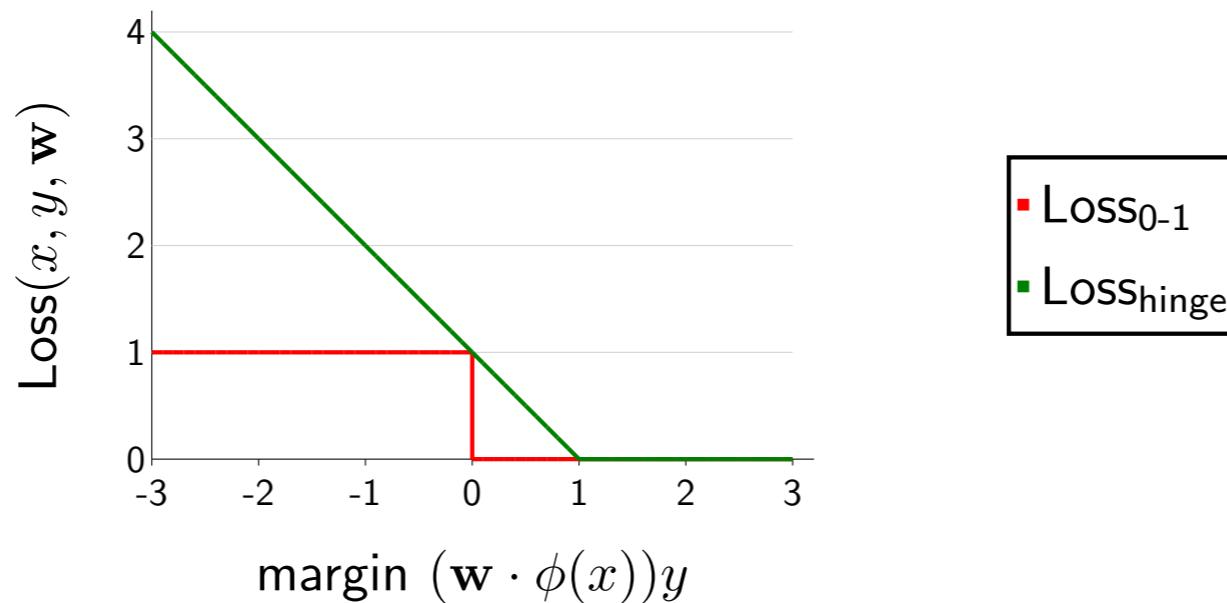
$$\nabla_{\mathbf{w}} \text{Loss}_{0-1}(x, y, \mathbf{w}) = \nabla \mathbf{1}[(\mathbf{w} \cdot \phi(x))y \leq 0]$$



Gradient is zero almost everywhere!

- Now we consider the third design decision, the optimization algorithm for minimizing the training loss.
- Let's just go with gradient descent. Recall that to run gradient descent, we need to first compute the gradient.
- The gradient of the training loss is just the average over the per-example losses. And then we need to take the gradient of indicator function...
- 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.
- If you try running gradient descent on a function with zero gradient, you will be stuck.
- One's first reaction to why the zero-one loss is hard to optimize is that it is not differentiable (everywhere). However, that is not really the real reason. The real reason is because it has zero gradients.

Hinge loss

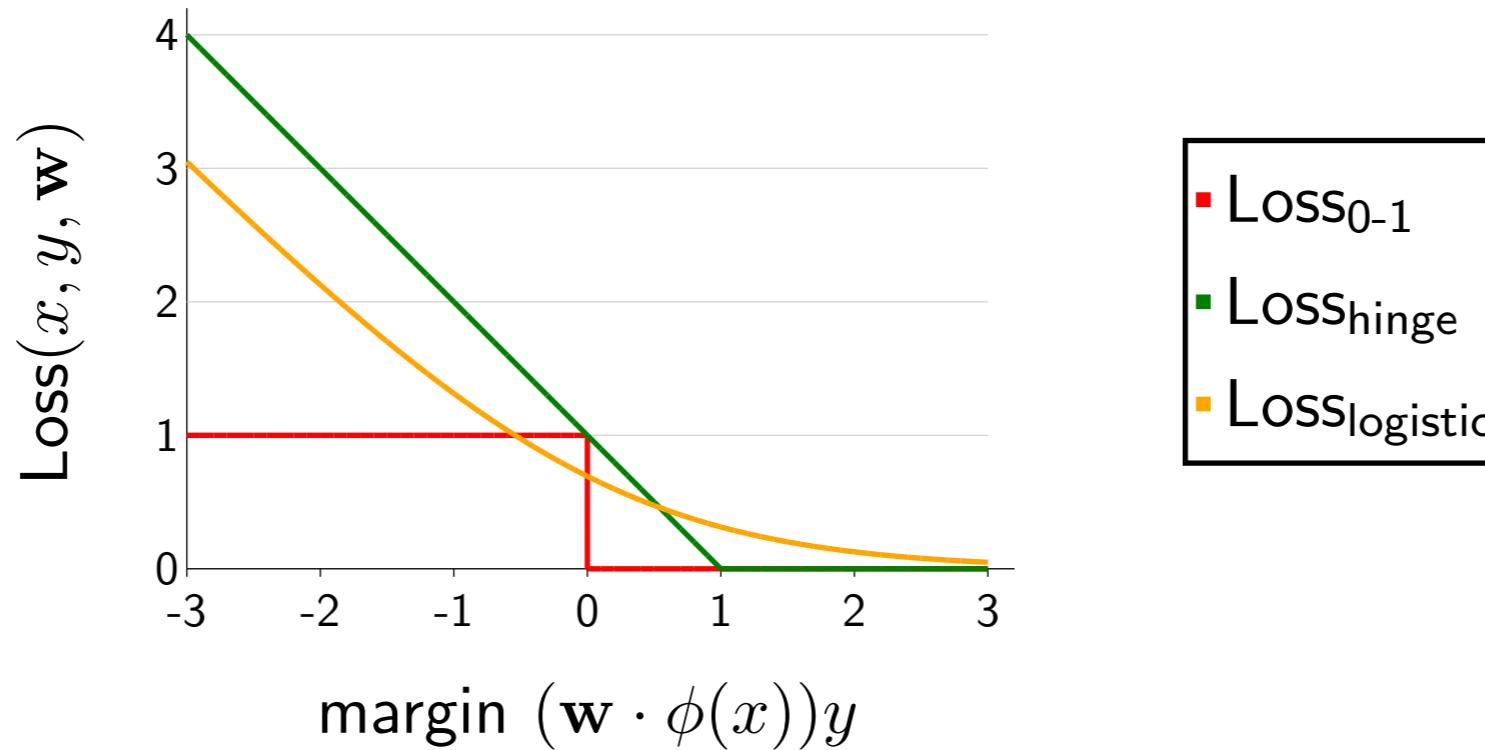


$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

- To fix this problem, we have to choose another loss function.
- A popular loss function is the **hinge loss**, which is the maximum over a descending line and the zero function. It is best explained visually.
- If the margin is at least 1, then the hinge loss is zero.
- If the margin is less than 1, then the hinge loss rises linearly.
- The 1 is there to provide some buffer: we ask the classifier to predict not only correctly, but by a (positive) margin of safety.
- Aside: Technically, the 1 can be any positive number. If we have regularization, it is equivalent to setting the regularization strength.
- Also note that the hinge loss is an upper bound on the zero-one loss, so driving down the hinge loss will generally drive down the zero-one loss. In particular, if the hinge loss is zero, the zero-one loss must also be zero.

Digression: logistic regression

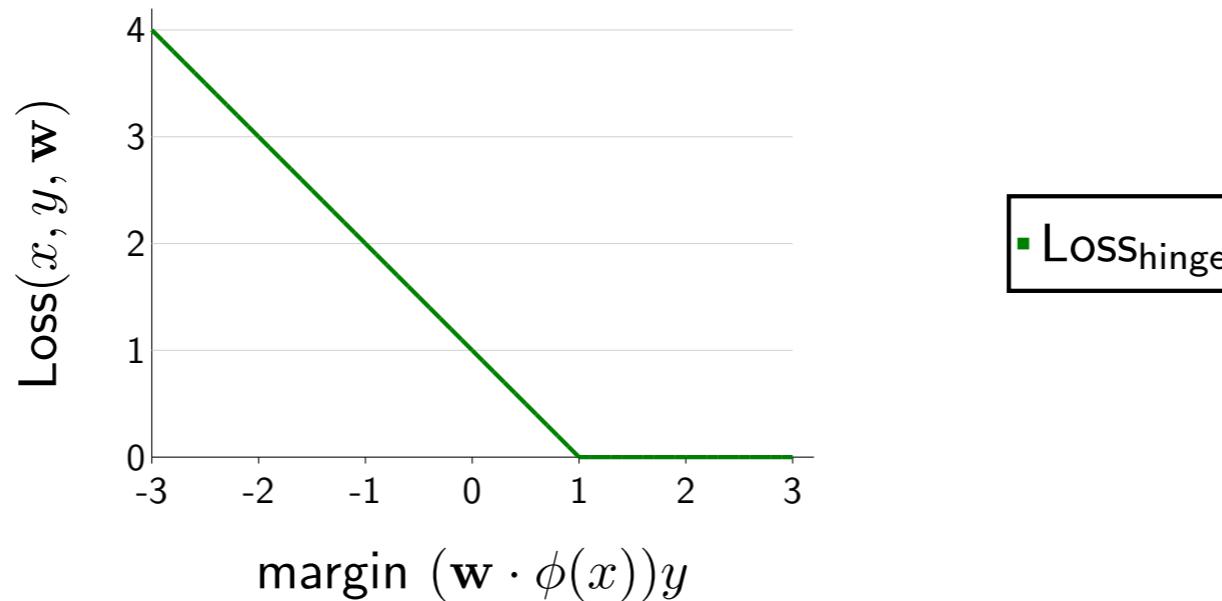
$$\text{LOSS}_{\text{logistic}}(x, y, \mathbf{w}) = \log(1 + e^{-(\mathbf{w} \cdot \phi(x))y})$$



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

- Another popular loss function used in machine learning 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



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

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

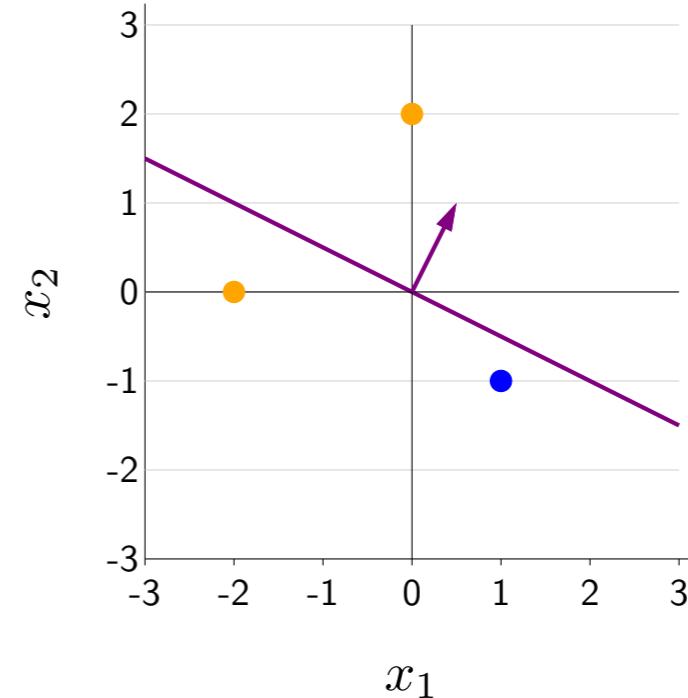
- You should try to "see" the solution before you write things down formally. Pictorially, it should be evident: when the margin is less than 1, then the gradient is the gradient of $1 - (\mathbf{w} \cdot \phi(x))y$, which is equal to $-\phi(x)y$. If the margin is larger than 1, then the gradient is 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. But in practice, you can take either $-\phi(x)y$ or 0.
- 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}' .

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



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

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

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

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

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

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

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

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

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

- 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, the loss is zero, and therefore the gradient is zero.
- For the second example, the loss is non-zero which is expected since the classifier is incorrect. The gradient is non-zero.
- For the third example, note that the loss is non-zero even though the classifier is correct. This is because we have to have a margin of 1, but the margin in this case is only 0.5.

Gradient descent (hinge loss) in Python

[code]

- Let us start from the regression code and change the loss function.
- Note that we don't have to modify the optimization algorithm at all, a benefit of decoupling the objective function from the optimization algorithm.



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 ($\text{score} - y$)	margin ($\text{score} y$)
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.



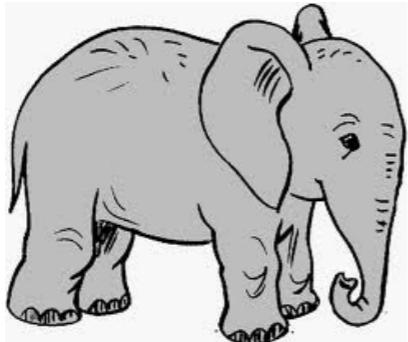
Machine learning: stochastic gradient descent



- In this module, we will introduce stochastic gradient descent.

Gradient descent is slow

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



Algorithm: gradient descent

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

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

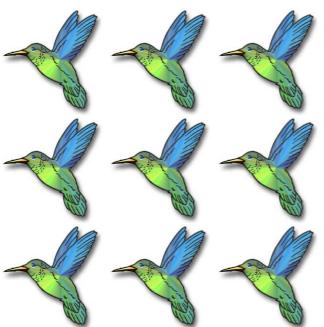
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

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.
- But one problem with gradient descent is that it is slow.
- Recall that the training loss is a sum over the training data. If we have one million training examples, 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?

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

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

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

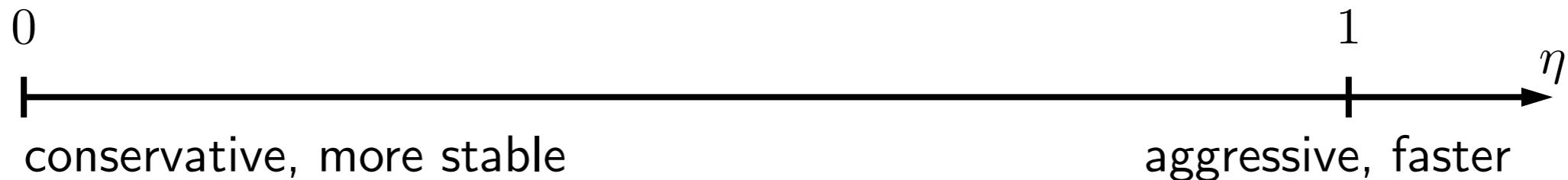
$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$

- The answer is **stochastic gradient descent** (SGD).
- Rather than looping through all the training examples to compute a single gradient and making one step, SGD loops through the examples (x, y) and updates the weights w based on **each** example.
- Each update is not as good because we're only looking at one example rather than all the examples, but we can make many more updates this way.
- Aside: there is a continuum between SGD and GD called minibatch SGD, where each update consists of an average over B examples.
- Aside: There are other variants of SGD. You can randomize the order in which you loop over the training data in each iteration. Think about why this is important if in your training data, you had 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 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 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.
- 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.
- Aside: There are some nice theoretical results showing that SGD is guaranteed to converge in this case (provided all your gradients are bounded).

Stochastic gradient descent in Python

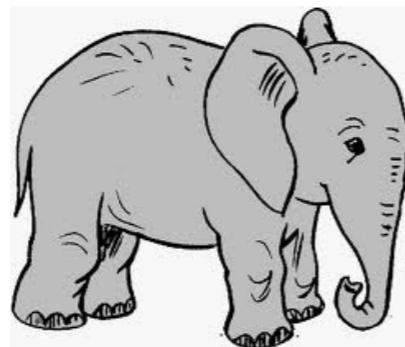
[code]

- Now let us code up stochastic gradient descent for linear regression in Python.
- First we generate a large enough dataset so that speed actually matters. We will also generate 1 million points according to $x \sim \mathcal{N}(0, I)$ and $y \sim \mathcal{N}(\mathbf{w}^* \cdot x, 1)$, where \mathbf{w}^* is the true weight vector, but hidden to the algorithm.
- This way, we can diagnose whether the algorithm is actually working or not by checking whether it recovers something close to \mathbf{w}^* .
- Let's first run gradient descent, and watch that it makes progress but it is very slow.
- Now let us implement stochastic gradient descent. It is much faster.

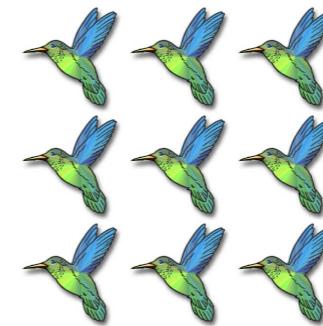


Summary

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



gradient descent



stochastic gradient descent



Key idea: stochastic updates

It's not about **quality**, it's about **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).
- Of course, sometimes stochastic gradient descent can be unstable, and other techniques such as mini-batching can be used to stabilize it.



Machine learning: 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.

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.

False arrest due to facial recognition



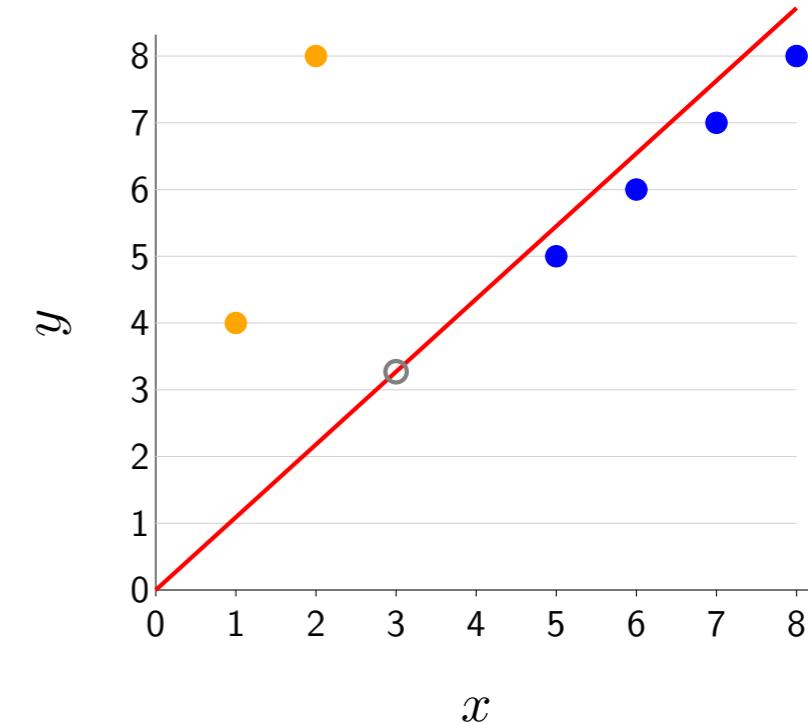
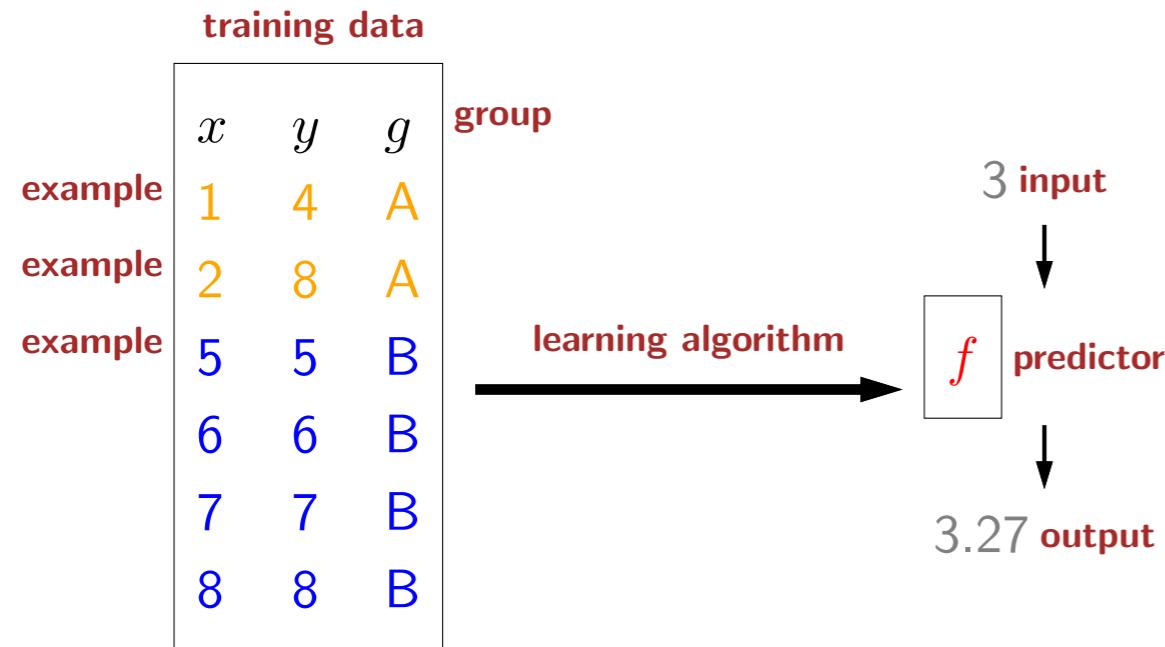
Wrongfully Accused by an Algorithm

In what may be the first known case of its kind, a faulty facial recognition match led to a Michigan man's arrest for a crime he did not commit.

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 an 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



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

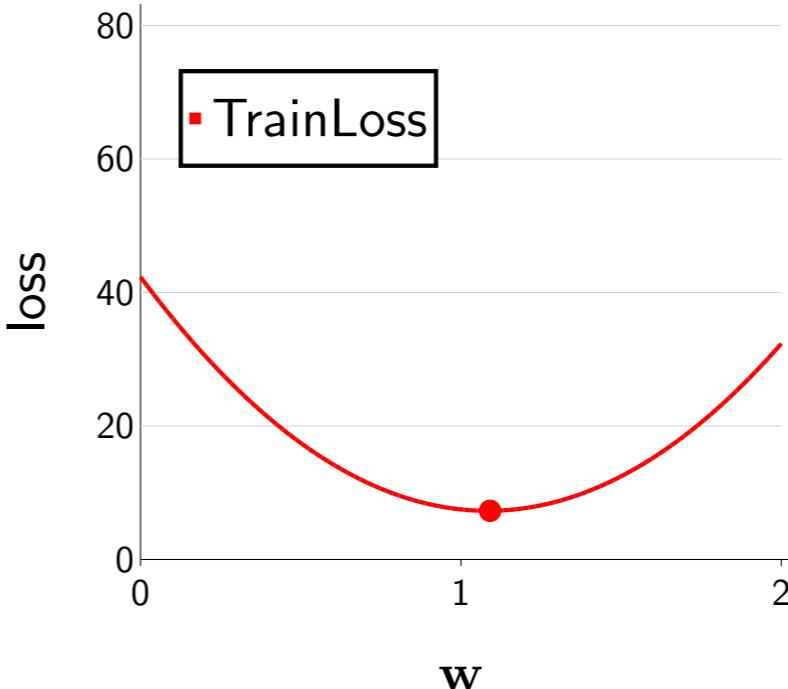
Note: predictor $f_{\mathbf{w}}$ does not use group information g

- 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_w computes the dot product of the weight vector 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_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



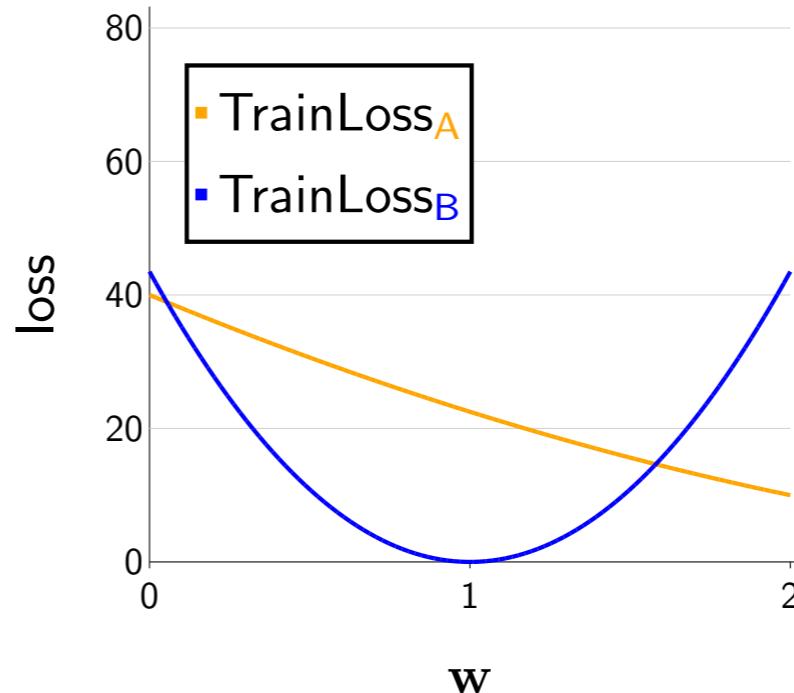
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

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

- 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



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

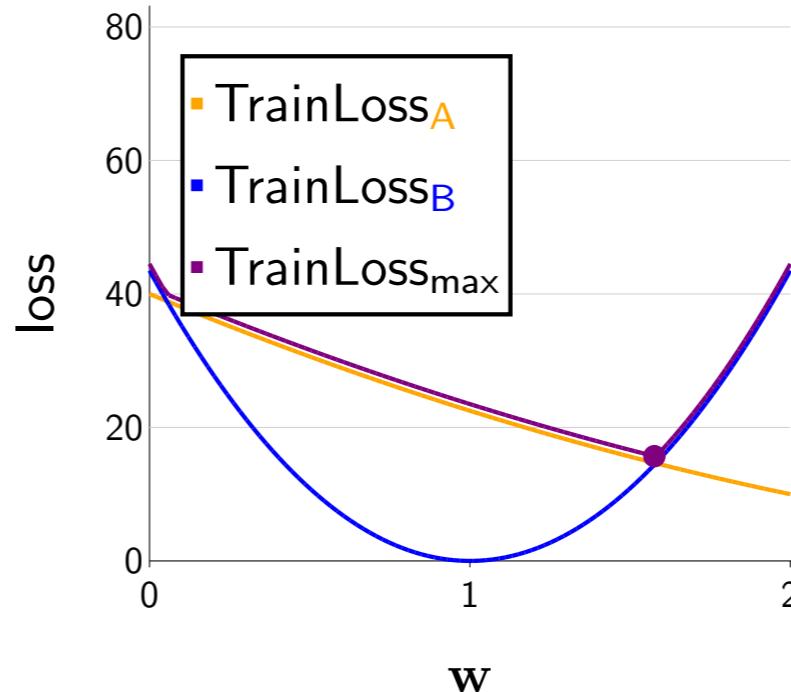
$$\text{TrainLoss}_A(1) = \frac{1}{2}((1 - 4)^2 + (2 - 8)^2) = 22.5$$

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

- 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



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

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

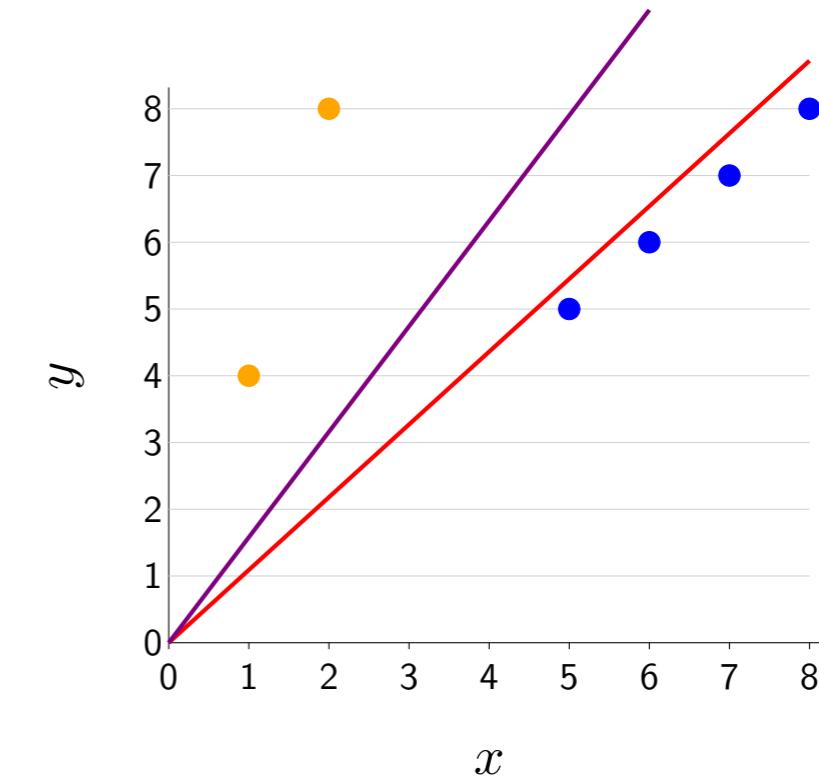
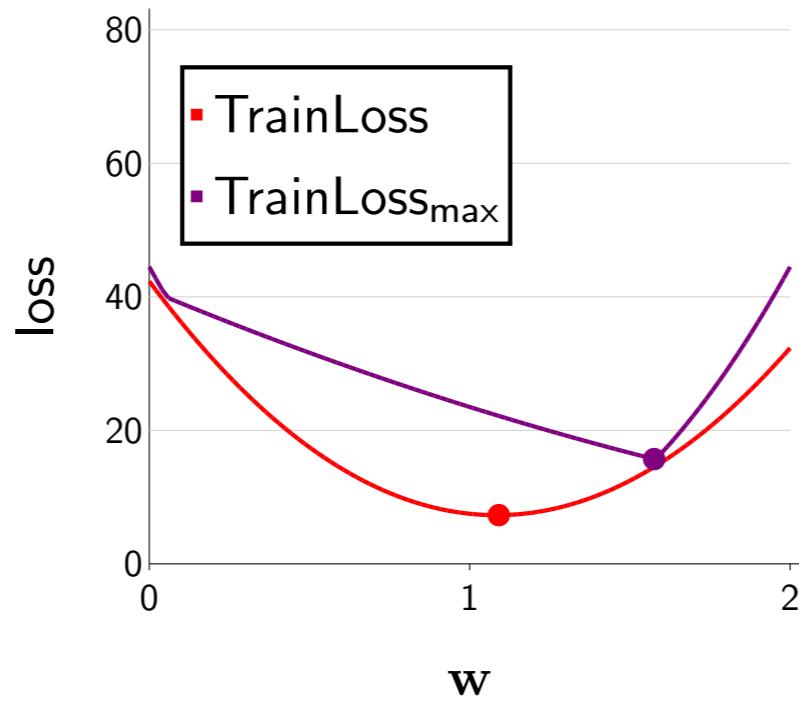
$$\text{TrainLoss}_B(1) = 0$$

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

- 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: $w = 1.09$

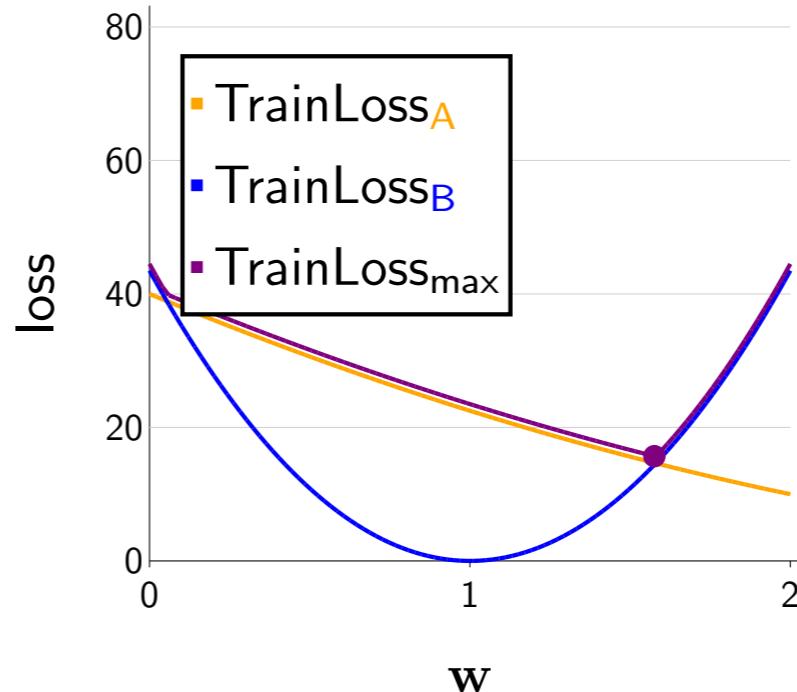
Group distributionally robust optimization (group DRO):

minimizer of maximum group loss: $w = 1.58$

- Let us now compare the old average loss (the standard training loss) TrainLoss and the new maximum group loss TrainLoss_{\max} .
- We minimize the average loss by setting the slope $w = [1.09]$, yielding an average loss of 7.29. However, this setting gets much higher maximum group loss (over 20). Pictorially, we see that this w heavily favors the majority group (B).
- If instead we minimize the maximum group loss directly, we would choose $w = 1.58$, 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



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

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

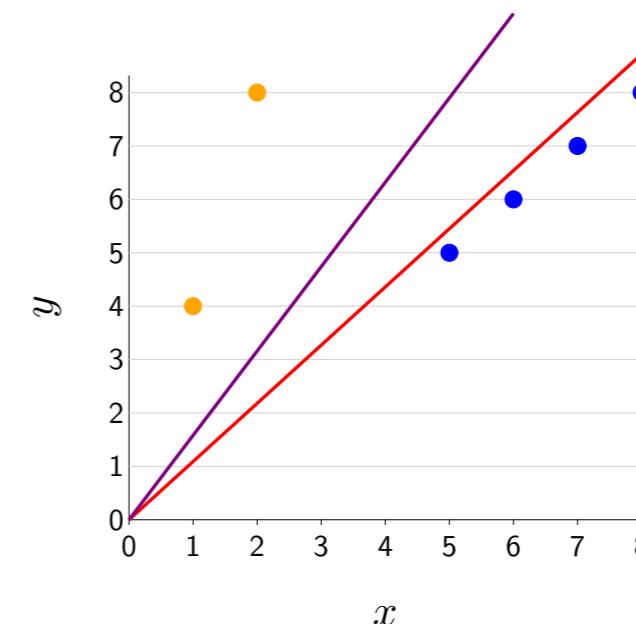
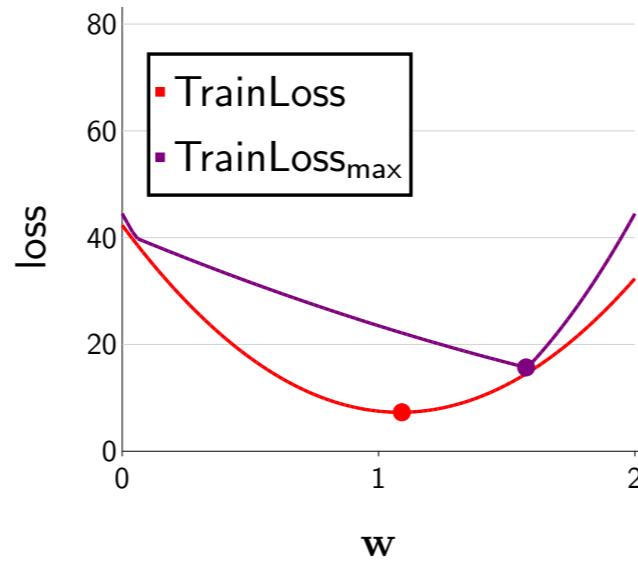
where $g^* = \arg \max_g \text{TrainLoss}_g(\mathbf{w})$

- In general, we can find minimize the maximum group loss by gradient descent.
- We just have to be able to take the gradient of TrainLoss_{\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

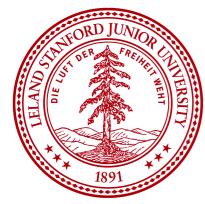


x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B



- 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 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,



Machine learning: non-linear features

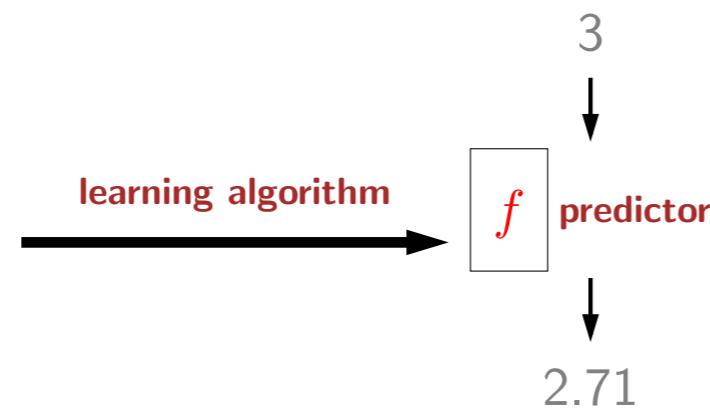


- In this module, we'll show that even using the machinery of **linear** models, we can obtain much more powerful **non-linear** predictors.

Linear regression

training data

x	y
1	1
2	3
4	3



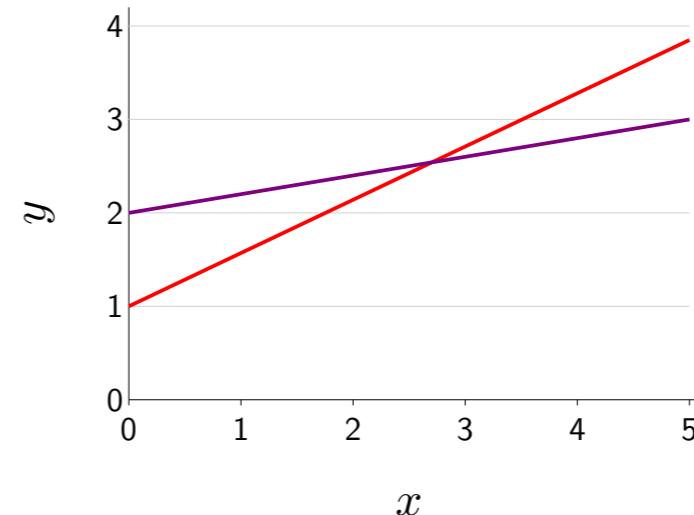
Which predictors are possible?
Hypothesis class

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^d\}$$

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

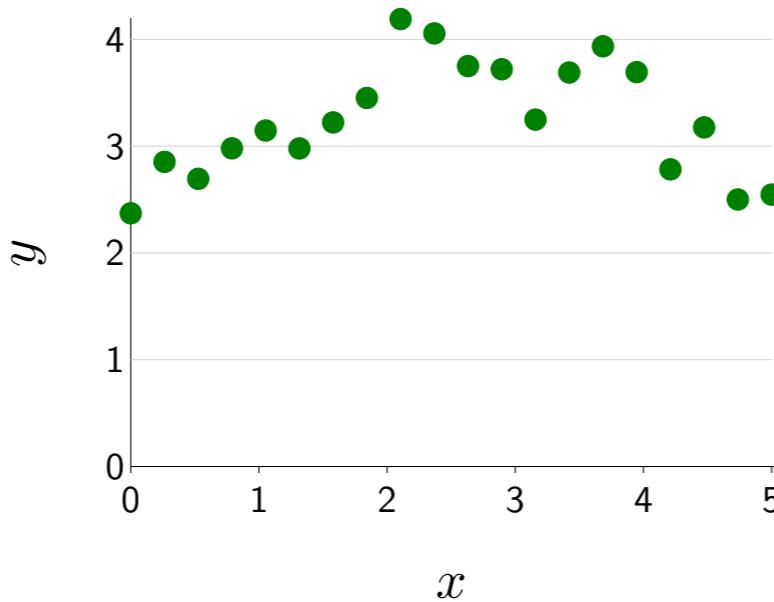
$$f(x) = [1, 0.57] \cdot \phi(x)$$

$$f(x) = [2, 0.2] \cdot \phi(x)$$



- We will look at regression and later turn to classification.
- Recall that in linear regression, given training data, a learning algorithm produces a predictor that maps new inputs to new outputs. The first design decision: what are the possible predictors that the learning algorithm can consider (what is the hypothesis class)?
- For linear predictors, remember the hypothesis class is the set of predictors that map some input x to the dot product between some weight vector w and the feature vector $\phi(x)$.
- As a simple example, if we define the feature extractor to be $\phi(x) = [1, x]$, then we can define various linear predictors with different intercepts and slopes.

More complex data



How do we fit a non-linear predictor?

- But sometimes data might be more complex and not be easily fit by a linear predictor. In this case, what can we do?
- One immediate reaction might be to go to something fancier like neural networks or decision trees.
- But let's see how far we can get with the machinery of linear predictors first.

Quadratic predictors

$$\phi(x) = [1, x, x^2]$$

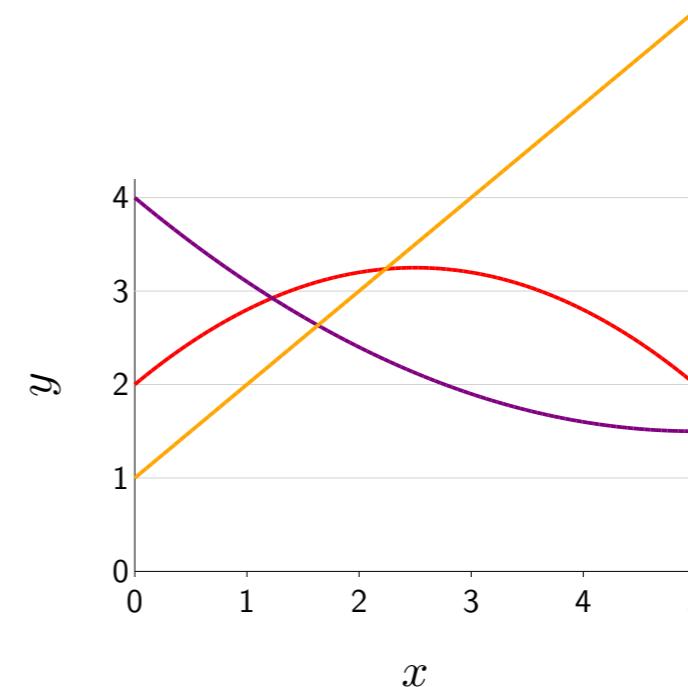
Example: $\phi(3) = [1, 3, 9]$

$$f(x) = [2, 1, -0.2] \cdot \phi(x)$$

$$f(x) = [4, -1, 0.1] \cdot \phi(x)$$

$$f(x) = [1, 1, 0] \cdot \phi(x)$$

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^3\}$$



Non-linear predictors just by changing ϕ

- The key observation is that the feature extractor ϕ can be arbitrary.
- So let us define it to include an x^2 term.
- Now, by setting the weights appropriately, we can define a non-linear (specifically, a **quadratic**) predictor.
- The first two examples of quadratic predictors vary in intercept, slope and curvature.
- Note that by setting the weight for feature x^2 to zero, we recover linear predictors.
- Again, the hypothesis class is the set of all predictors f_w obtained by varying w .
- Note that the hypothesis class of quadratic predictors is a **superset** of the hypothesis class of linear predictors.
- In summary, we've seen our first example of obtaining non-linear predictors just by changing the feature extractor ϕ !
- Advanced: here $x \in \mathbb{R}$ is one-dimensional, so x^2 is just one additional feature. If $x \in \mathbb{R}^d$ were d -dimensional, then there would be $O(d^2)$ quadratic features of the form $x_i x_j$ for $i, j \in \{1, \dots, d\}$. When d is large, then d^2 can be prohibitively large, which is one reason that using the machinery of linear predictors to increase expressivity can be problematic.

Piecewise constant predictors

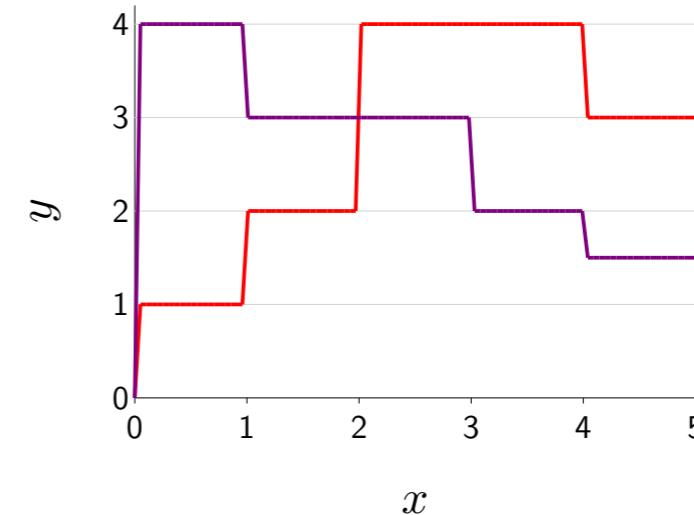
$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \mathbf{1}[2 < x \leq 3], \mathbf{1}[3 < x \leq 4], \mathbf{1}[4 < x \leq 5]]$$

Example: $\phi(2.3) = [0, 0, 1, 0, 0]$

$$f(x) = [1, 2, 4, 4, 3] \cdot \phi(x)$$

$$f(x) = [4, 3, 3, 2, 1.5] \cdot \phi(x)$$

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^5\}$$



Expressive non-linear predictors by partitioning the input space

- Quadratic predictors are still a bit restricted: they can only go up and then down smoothly (or vice-versa).
- We introduce another type of feature extractor which divides the input space into regions and allows the predicted value of each region to vary independently, yielding piecewise constant predictors (see figure).
- Specifically, each component of the feature vector corresponds to one region (e.g., $[0, 1)$) and is 1 if x lies in that region and 0 otherwise.
- Assuming the regions are disjoint, the weight associated with a component/region is exactly the predicted value.
- As you make the regions smaller, then you have more features, and the expressivity of your hypothesis class increases. In the limit, you can essentially capture any predictor you want.
- Advanced: what happens if x were not a scalar, but a d -dimensional vector? Then if each component gets broken up into B bins, then there will be B^d features! For each feature, we need to fit its weight, and there will in generally be too few examples to fit all the features.

Predictors with periodicity structure

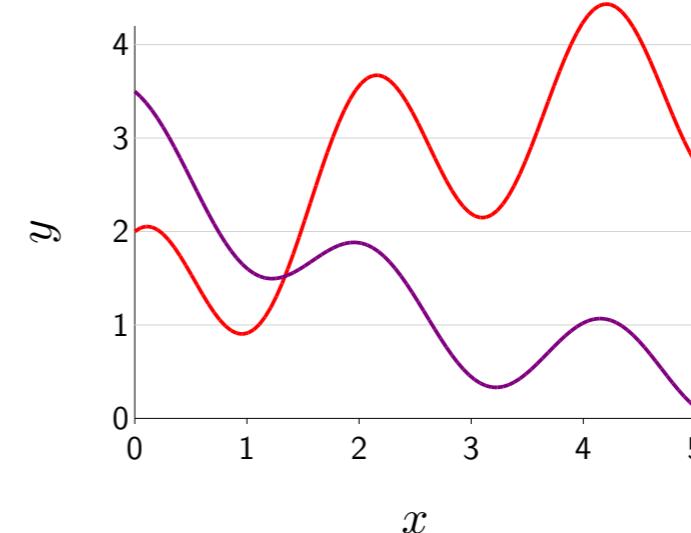
$$\phi(x) = [1, x, x^2, \cos(3x)]$$

Example: $\phi(2) = [1, 2, 4, 0.96]$

$$f(x) = [1, 1, -0.1, 1] \cdot \phi(x)$$

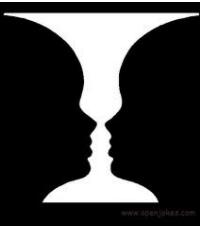
$$f(x) = [3, -1, 0.1, 0.5] \cdot \phi(x)$$

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^4\}$$



Just throw in any features you want

- Quadratic and piecewise constant predictors are just two examples of an unboundedly large design space of possible feature extractors.
- Generally, the choice of features is informed by the prediction task that we wish to solve (either prior knowledge or preliminary data exploration).
- For example, if x represents time and we believe the true output y varies according to some periodic structure (e.g., traffic patterns repeat daily, sales patterns repeat annually), then we might use periodic features such as cosine to capture these trends.
- Each feature might represent some type of structure in the data. If we have multiple types of structures, these can just be "thrown in" into the feature vector.
- Features represent what properties **might** be useful for prediction. If a feature is not useful, then the learning algorithm can assign a weight close to zero to that feature. Of course, the more features one has, the harder learning becomes.



Linear in what?

Prediction:

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

Linear in \mathbf{w} ? Yes

Linear in $\phi(x)$? Yes

Linear in x ? No!



Key idea: non-linearity

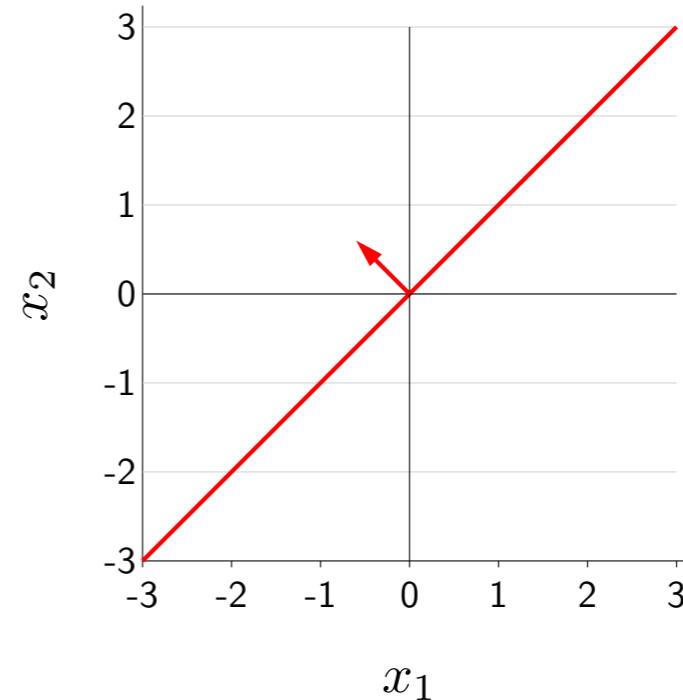
- Expressivity: score $\mathbf{w} \cdot \phi(x)$ can be a **non-linear** function of x
- Efficiency: score $\mathbf{w} \cdot \phi(x)$ always a **linear** function of \mathbf{w}

- Wait a minute...how are we able to obtain non-linear predictors if we're still using the machinery of linear predictors? It's a linguistic sleight of hand, as "linear" is ambiguous.
- The score is $\mathbf{w} \cdot \phi(x)$ linear in \mathbf{w} and $\phi(x)$. However, the score is not linear in x (it might not even make sense because x need not be a vector at all — it could be a string or a PDF file).
- The significance is as follows: From the feature extractor's viewpoint, we can define arbitrary features that yield very **non-linear** functions in x .
- From the learning algorithm's viewpoint (which only looks at $\phi(x)$, not x), **linearity** us to optimize the weights efficiently.
- Advanced: if the score is linear in \mathbf{w} and the loss function Loss is convex (which holds for the squared, hinge, logistic losses but not the zero-one loss), then minimizing the training loss TrainLoss is a convex optimization problem, and gradient descent with a proper step size is guaranteed to converge to the global minimum.

Linear classification

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

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



Decision boundary is a line

- Now let's turn from regression to classification.
- The story is pretty much the same: you can define arbitrary features to yield non-linear classifiers.
- Recall that in binary classification, the classifier (predictor) returns the sign of the score.
- The classifier can therefore be represented by its decision boundary, which divides the input space into two regions: points with positive score and points with negative score.
- Note that the classifier $f_w(x)$ is a non-linear function of x (and $\phi(x)$) no matter what (due to the sign function), so it is not helpful to talk about whether f_w is linear or non-linear. Instead we will ask whether the **decision boundary** corresponding to f_w is linear or not.

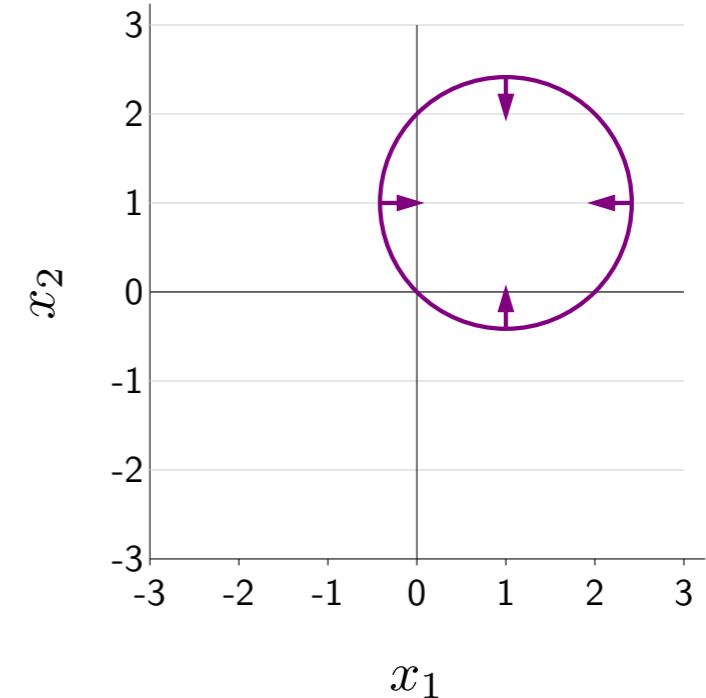
Quadratic classifiers

$$\phi(x) = [x_1, x_2, x_1^2 + x_2^2]$$

$$f(x) = \text{sign}([2, 2, -1] \cdot \phi(x))$$

Equivalently:

$$f(x) = \begin{cases} 1 & \text{if } \{(x_{-1} - 1)^2 + (x_{-2} - 1)^2 \leq 2\} \\ -1 & \text{otherwise} \end{cases}$$



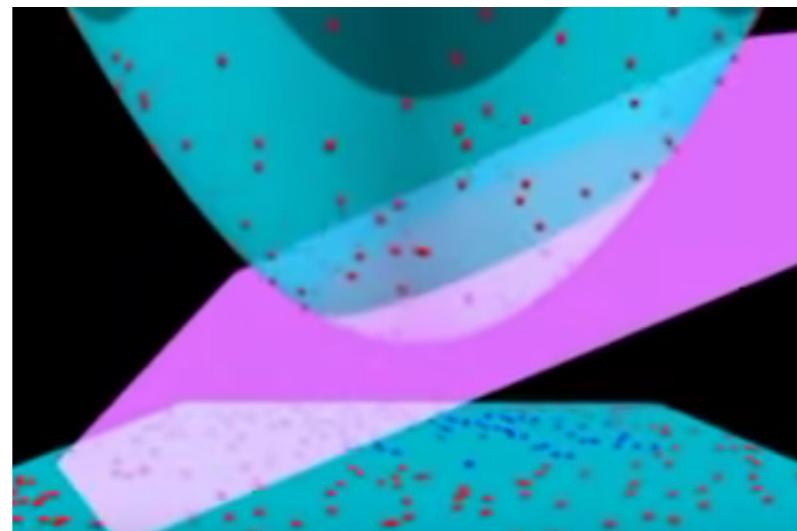
Decision boundary is a circle

- Let us see how we can define a classifier with a non-linear decision boundary.
- Let's try to construct a feature extractor that induces a decision boundary that is a circle: the inside is classified +1 and the outside is classified -1.
- We will add a new feature $x_1^2 + x_2^2$ into the feature vector, and define the weights to be as follows.
- Then rewrite the classifier to make it clear that it is the equation for the interior of a circle with radius $\sqrt{2}$.
- As a sanity check, we you can see that $x = [0, 0]$ results in a score of 0, which means that it is on the decision boundary. And as either of x_1 or x_2 grow in magnitude (either $|x_1| \rightarrow \infty$ or $|x_2| \rightarrow \infty$), the contribution of the third feature dominates and the sign of the score will be negative.

Visualization in feature space

Input space: $x = [x_1, x_2]$, decision boundary is a circle

Feature space: $\phi(x) = [x_1, x_2, x_1^2 + x_2^2]$, decision boundary is a line



- Let's try to understand the relationship between the non-linearity in x and linearity in $\phi(x)$.
- Click on the image to see the linked video (which is about polynomial kernels and SVMs, but the same principle applies here).
- In the input space x , the decision boundary which separates the red and blue points is a circle.
- We can also visualize the points in **feature space**, where each point is given an additional dimension $x_1^2 + x_2^2$.
- In this three-dimensional feature space, a linear predictor (which is now defined by a hyperplane instead of a line) can in fact separate the red and blue points.
- This corresponds to the non-linear predictor in the original two-dimensional space.

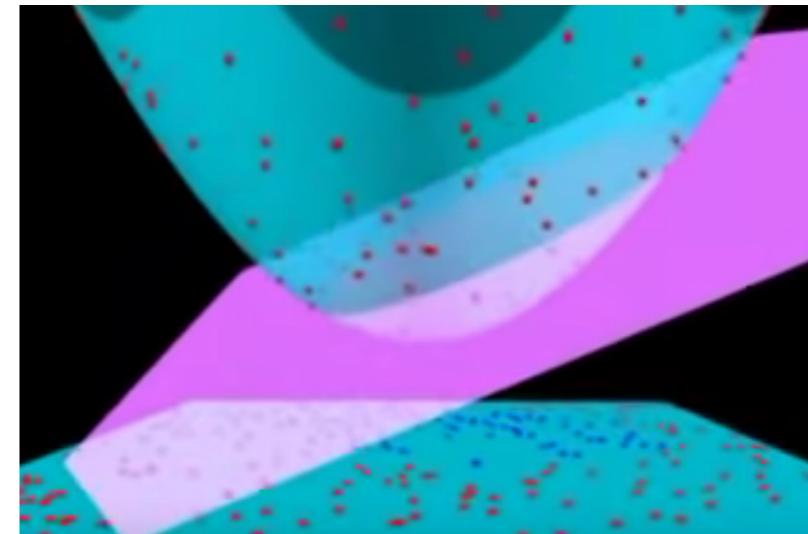


Summary

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

linear in $\mathbf{w}, \phi(x)$

non-linear in x



- Regression: non-linear predictor, classification: non-linear decision boundary
- Types of non-linear features: quadratic, piecewise constant, etc.

Non-linear predictors with linear machinery

- To summarize, we have shown that the term "linear" is ambiguous: a predictor in regression is non-linear in the input x but is linear in the feature vector $\phi(x)$.
- The score is also linear with respect to the weights w , which is important for efficient learning.
- Classification is similar, except we talk about (non-)linearity of the decision boundary.
- We also saw many types of non-linear predictors that you could create by concocting various features (quadratic predictors, piecewise constant predictors).
- So next time someone on the street asks you about linear predictors, you should first ask them "linear in what?"



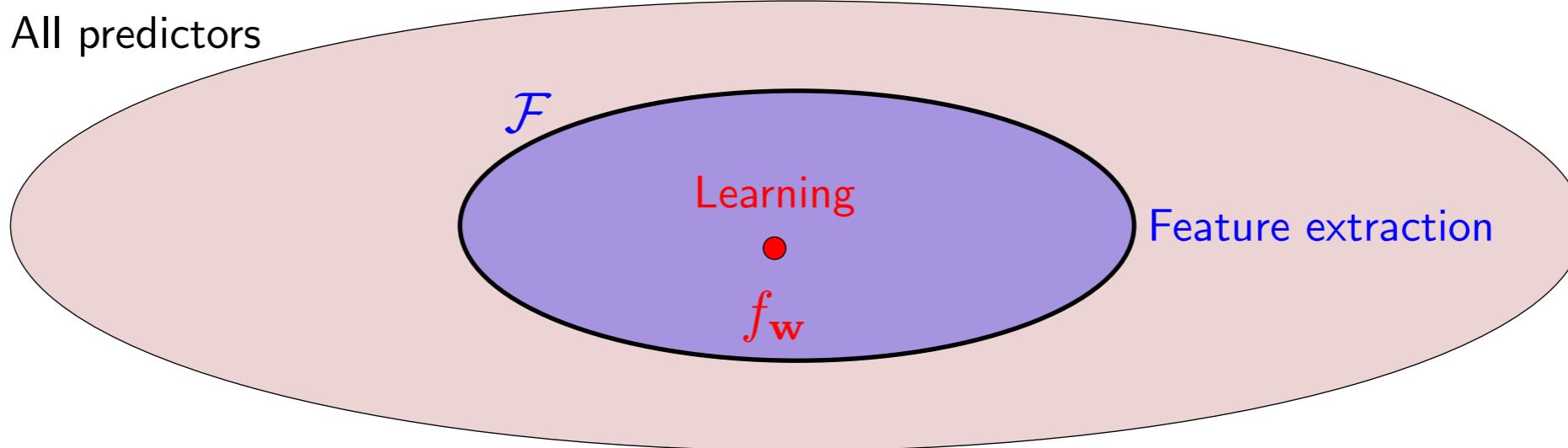
Machine learning: feature templates



- In this module, we'll talk about how to use feature templates to construct features in a flexible way.

Feature extraction + learning

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$



- Feature extraction: choose \mathcal{F} based on domain knowledge
- Learning: choose $f_{\mathbf{w}} \in \mathcal{F}$ based on data

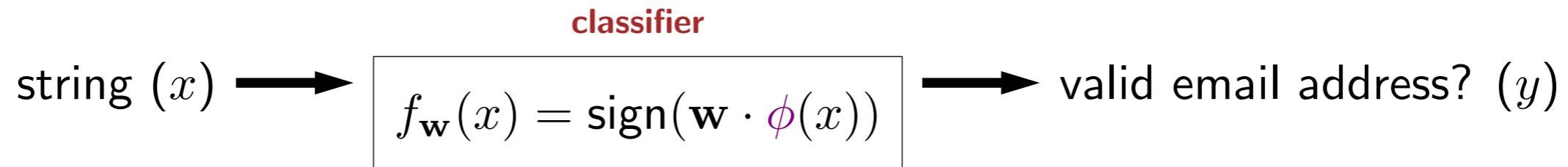
Want \mathcal{F} to contain good predictors but not be too big

- Recall that the hypothesis class \mathcal{F} is the set of predictors considered by the learning algorithm. In the case of linear predictors, \mathcal{F} is given by some function of $\mathbf{w} \cdot \phi(x)$ for all \mathbf{w} (sign for classification, no sign for regression). This can be visualized as a set in the figure.
- Learning is the process of choosing a particular predictor $f_{\mathbf{w}}$ from \mathcal{F} given training data.
- But the question that will concern us in this module is how do we choose \mathcal{F} ? We saw some options already: linear predictors, quadratic predictors, etc., but what makes sense for a given application?
- If the hypothesis class doesn't contain any good predictors, then no amount of learning can help. So the question when extracting features is really whether they are powerful enough to **express** good predictors. It's okay and expected that \mathcal{F} will contain bad ones as well. Of course, we don't want \mathcal{F} to be too big, or else learning becomes hard, not just computationally but statistically (as we'll explain when we talk about generalization).



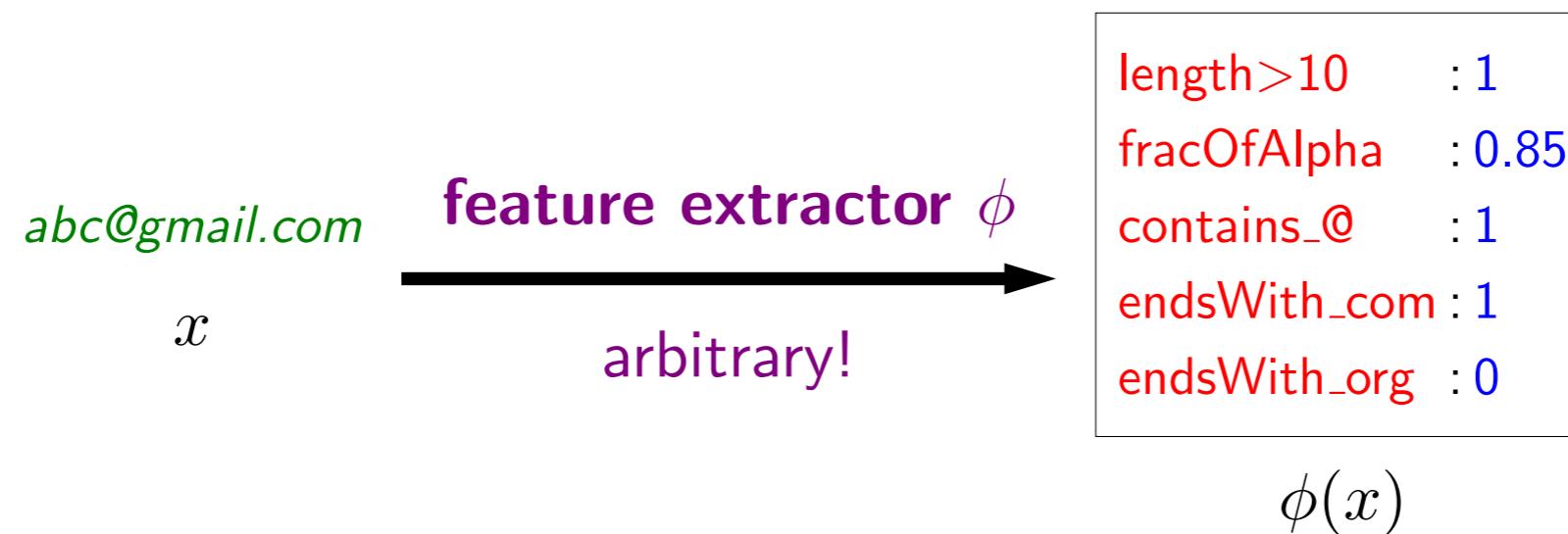
Feature extraction with feature names

Example task:



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

Feature extractor: Given x , produce set of (feature name, feature value) pairs



- To get some intuition about feature extraction, let us consider the task of predicting whether a string is a valid email address or not.
- We will assume the classifier f_w is a linear classifier, which is given by some feature extractor ϕ .
- Feature extraction is a bit of an art that requires intuition about both the task and also what machine learning algorithms are capable of. The general principle is that features should represent properties of x which **might be** relevant for predicting y .
- Think about the feature extractor as producing a set of (feature name, feature value) pairs. For example, we might extract information about the length, or fraction of alphanumeric characters, whether it contains various substrings, etc.
- It is okay to add features which turn out to be irrelevant, since the learning algorithm can always in principle choose to ignore the feature, though it might take more data to do so.
- We have been associating each feature with a name so that it's easier for us (humans) to interpret and develop the feature extractor. The feature names act like the analogue of **comments** in code. Mathematically, the feature name is not needed by the learning algorithm and erasing them does not change prediction or learning.

Prediction with feature names

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

length>10	:-1.2
fracOfAlpha	:0.6
contains_@	:3
endsWith_com	:2.2
endsWith_org	:1.4

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

length>10	:1
fracOfAlpha	:0.85
contains_@	:1
endsWith_com	:1
endsWith_org	:0

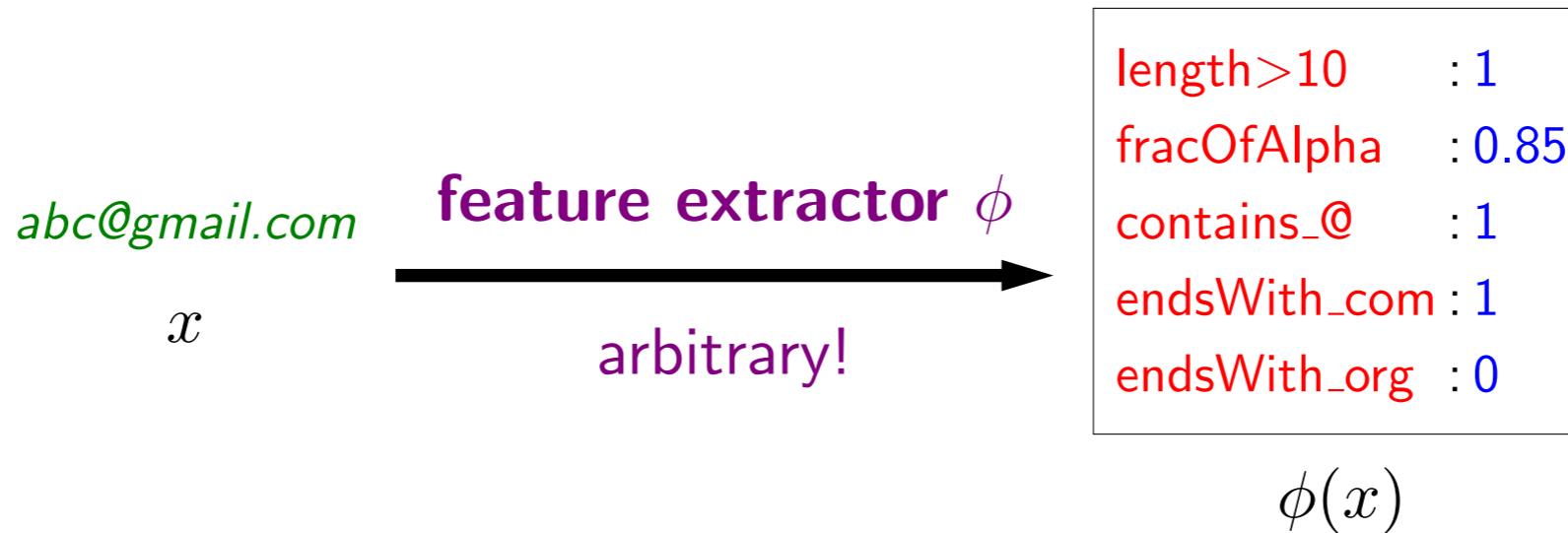
Score: weighted combination of features

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

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

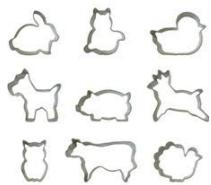
- A feature vector formally is just a list of numbers, but we have endowed each feature in the feature vector with a name.
- The weight vector is also just a list of numbers, but we can endow each weight with the corresponding name as well.
- Recall that the score is simply the dot product between the weight vector and the feature vector. In other words, the score aggregates the contribution of each feature, weighted appropriately.
- Each feature weight w_j determines how the corresponding feature value $\phi_j(x)$ contributes to the prediction.
- If w_j is positive, then the presence of feature j ($\phi_j(x) = 1$) favors a positive classification (e.g., ending with com). Conversely, if w_j is negative, then the presence of feature j favors a negative classification (e.g., length greater than 10). The magnitude of w_j measures the strength or importance of this contribution.
- Advanced: while tempting, it can be a bit misleading to interpret feature weights in isolation, because the learning algorithm treats \mathbf{w} holistically. In particular, a feature weight w_j produced by a learning algorithm will change depending on the presence of other features. If the weight of a feature is positive, it doesn't necessarily mean that feature is positively correlated with the label.

Organization of features?

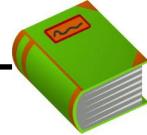


Which features to include? Need an organizational principle...

- How would we go about creating good features?
- Here, we used our prior knowledge to define certain features (`contains_@`) which we believe are helpful for detecting email addresses.
- But this is ad-hoc, and it's easy to miss useful features (e.g., `endsWith_us`), and there might be other features which are predictive but not intuitive.
- We need a more systematic way to go about this.



Feature templates



Definition: feature template

A **feature template** is a group of features all computed in a similar way.

abc@gmail.com

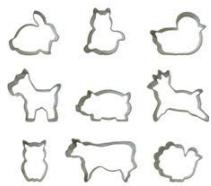


last three characters equals ___

endsWith_aaa : 0
endsWith_aab : 0
endsWith_aac : 0
...
endsWith_com : 1
...
endsWith_zzz : 0

Define types of pattern to look for, not particular patterns

- A useful organization principle is a **feature template**, which groups all the features which are computed in a similar way. (People often use the word "feature" when they really mean "feature template".)
- Rather than defining individual features like `endsWith_com`, we can define a single feature template which expands into all the features that computes whether the input x matches any three characters.
- Typically, we will write a feature template as an English description with a blank (`_`), which is to be filled in with an arbitrary value.
- The upshot is that we don't need to know which particular patterns (e.g., three-character suffixes) are useful, but only that **existence** of certain patterns (e.g., three-character suffixes) are useful cue to look at.
- It is then up to the learning algorithm to figure out which patterns are useful by assigning the appropriate feature weights.



Feature templates example 1

Input:

abc@gmail.com

Feature template

Last three characters equals ___

Length greater than ___

Fraction of alphanumeric characters

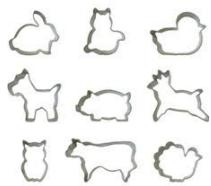
Example feature

Last three characters equals *com* : 1

Length greater than *10* : 1

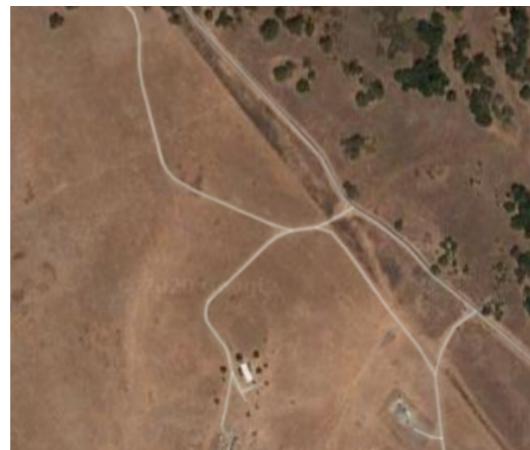
Fraction of alphanumeric characters : 0.85

- Here are some other examples of feature templates.
- Note that an isolated feature (e.g., fraction of alphanumeric characters) can be treated as a trivial feature template with no blanks to be filled.
- In many cases, the feature value is binary (0 or 1), but they can also be real numbers.



Feature templates example 2

Input:



Latitude: 37.4068176
Longitude: -122.1715122

Feature template

Pixel intensity of image at row ___ and column ___ (___ channel)

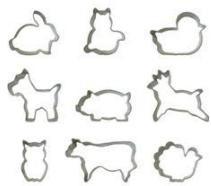
Latitude is in [___, ___] and longitude is in [___, ___]

Example feature name

Pixel intensity of image at row **10** and column **93** (**red** channel) : 0.8

Latitude is in [**37.4**, **37.5**] and longitude is in [**-122.2**, **-122.1**] : 1

- As another example application, suppose the input is an aerial image along with the latitude/longitude corresponding where the image was taken. This type of input arises in poverty mapping and land cover classification.
- In this case, we might define one feature template corresponding to the pixel intensities at various pixel-wise row/column positions in the image across all the 3 color channels (e.g., red, green, blue).
- Another feature template might define a family of binary features, one for each region of the world, where each region is defined by a bounding box over latitude and longitude.



Sparsity in feature vectors

abc@gmail.com

last character equals __

endsWith_a	: 0
endsWith_b	: 0
endsWith_c	: 0
endsWith_d	: 0
endsWith_e	: 0
endsWith_f	: 0
endsWith_g	: 0
endsWith_h	: 0
endsWith_i	: 0
endsWith_j	: 0
endsWith_k	: 0
endsWith_l	: 0
endsWith_m	: 1
endsWith_n	: 0
endsWith_o	: 0
endsWith_p	: 0
endsWith_q	: 0
endsWith_r	: 0
endsWith_s	: 0
endsWith_t	: 0
endsWith_u	: 0
endsWith_v	: 0
endsWith_w	: 0
endsWith_x	: 0
endsWith_y	: 0
endsWith_z	: 0

Compact representation:

{"endsWith_m": 1}

- In general, a feature template corresponds to many features, and sometimes, **for a given input**, most of the feature values are zero; that is, the feature vector is **sparse**.
- Of course, different feature vectors have different non-zero features.
- In this case, it would be inefficient to represent all the features explicitly. Instead, we can just store the values of the non-zero features, assuming all other feature values are zero by default.

Two feature vector implementations

Arrays (good for dense features):

```
pixelIntensity(0,0) : 0.8  
pixelIntensity(0,1) : 0.6  
pixelIntensity(0,2) : 0.5  
pixelIntensity(1,0) : 0.5  
pixelIntensity(1,1) : 0.8  
pixelIntensity(1,2) : 0.7  
pixelIntensity(2,0) : 0.2  
pixelIntensity(2,1) : 0  
pixelIntensity(2,2) : 0.1
```

[0.8, 0.6, 0.5, 0.5, 0.8, 0.7, 0.2, 0, 0.1]

Dictionaries (good for sparse features):

```
fracOfAlpha : 0.85  
contains_a : 0  
contains_b : 0  
contains_c : 0  
contains_d : 0  
contains_e : 0  
...  
contains_@ : 1  
...
```

{"fracOfAlpha": 0.85, "contains_@": 1}

- In general, there are two common ways to implement feature vectors: using arrays and using dictionaries.
- **Arrays** assume a fixed ordering of the features and store the feature values as an array. This implementation is appropriate when the number of nonzeros is significant (the features are dense). Arrays are especially efficient in terms of space and speed (and you can take advantage of GPUs). In computer vision applications, features (e.g., the pixel intensity features) are generally dense, so arrays are more common.
- However, when we have sparsity (few nonzeros), it is typically more efficient to implement the feature vector as a **dictionary** (map) from strings to doubles rather than a fixed-size array of doubles. The features not in the dictionary implicitly have a default value of zero. This sparse implementation is useful for natural language processing with linear predictors, and is what allows us to work efficiently over millions of features. In Python, one would define a feature vector $\phi(x)$ as the dictionary `{"endsWith_"+x[-3:]: 1}`. Dictionaries do incur extra overhead compared to arrays, and therefore dictionaries are much slower when the features are not sparse.
- One advantage of the sparse feature implementation is that you don't have to instantiate all the set of possible features in advance; the weight vector can be initialized to `{}`, and only when a feature weight becomes non-zero do we store it. This means we can dynamically update a model with incrementally arriving data, which might instantiate new features.



Summary

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$

Feature template:

abc@gmail.com

last three characters equals ___

endsWith_aaa : 0
endsWith_aab : 0
endsWith_aac : 0
...
endsWith_com : 1
...
endsWith_zzz : 0

Dictionary implementation:

{"endsWith_com": 1}

- The question we are concerned with in this module is to how to define the hypothesis class \mathcal{F} , which in the case of linear predictors is the question of what the feature extractor ϕ is.
- We showed how **feature templates** can be useful for organizing the definition of many features, and that we can use dictionaries to represent **sparse** feature vectors efficiently.
- Stepping back, feature engineering is one of the most critical components in the practice of machine learning. It often does not get as much attention as it deserves, mostly because it is a bit of an art and somewhat domain-specific.
- More powerful predictors such as neural networks will alleviate some of the burden of feature engineering, but even neural networks use feature vectors as the initial starting point, and therefore its effectiveness is ultimately governed by how good the features are.



Machine learning: neural networks

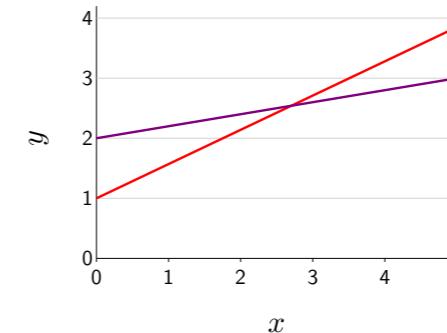


- In this module, I will present neural networks, a way to construct non-linear predictors via problem decomposition.

Non-linear predictors

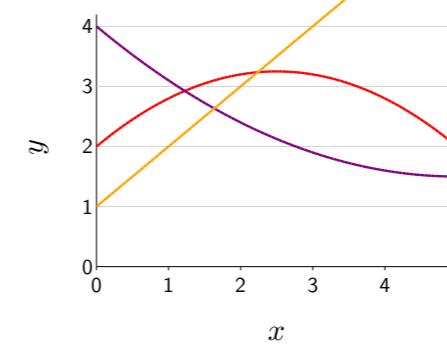
Linear predictors:

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



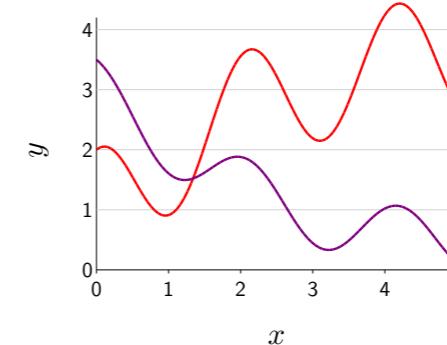
Non-linear (quadratic) predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x), \phi(x) = [1, x, x^2]$$



Non-linear neural networks:

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



- Recall that our first hypothesis class was linear (in x) predictors, which for regression means that the predictors are lines.
- However, we also showed that you could get non-linear (in x) predictors by simply changing the feature extractor ϕ . For example, by adding the feature x^2 , one obtains quadratic predictors.
- One disadvantage of this approach is that if x were d -dimensional, one would need $O(d^2)$ features and corresponding weights, which presents considerable computational and statistical challenges.
- We will show that with neural networks, we can leave the feature extractor alone, but increase the complexity of predictor, which can also produce non-linear (though not necessarily quadratic) predictors.
- It is a common misconception that neural networks allow you to express more complex predictors. You can define ϕ to include essentially all predictors (as is done in kernel methods).
- Rather, neural networks yield non-linear predictors in a more **compact** way. For instance, you might not need $O(d^2)$ features to represent the desired non-linear predictor.

Motivating example



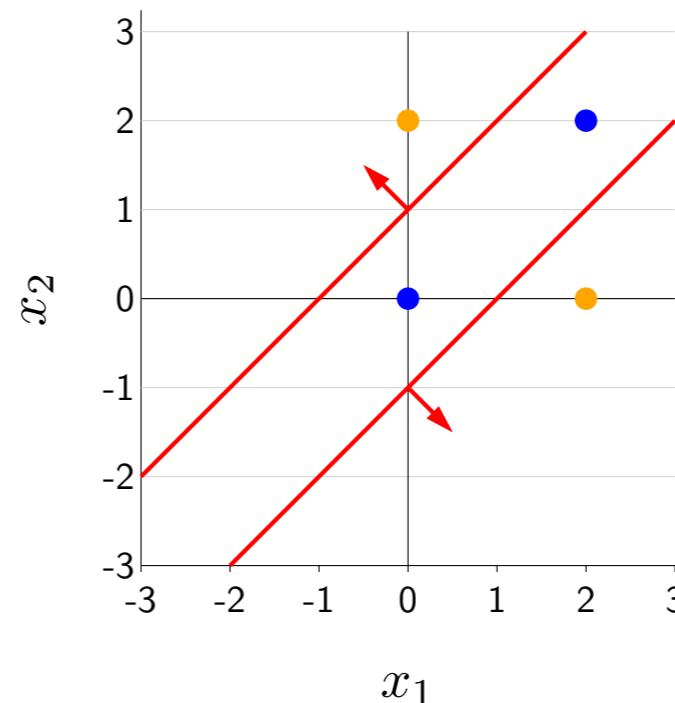
Example: predicting car collision

Input: positions of two oncoming cars $x = [x_1, x_2]$

Output: whether safe ($y = +1$) or collide ($y = -1$)

Unknown: safe if cars sufficiently far: $y = \text{sign}(|x_1 - x_2| - 1)$

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



- As a motivating example, consider the problem of predicting whether two cars are going to collide given the their positions (as measured from distance from one side of the road). In particular, let x_1 be the position of one car and x_2 be the position of the other car.
- Suppose the true output is 1 (safe) whenever the cars are separated by a distance of at least 1. This relationship can be represented by the decision boundary which labels all points in the interior region between the two red lines as negative, and everything on the exterior (on either side) as positive. Of course, this true input-output relationship is unknown to the learning algorithm, which only sees training data. Consider a simple training dataset consisting of four points. (This is essentially the famous XOR problem that was impossible to fit using linear classifiers.)

Decomposing the problem

Test if car 1 is far right of car 2:

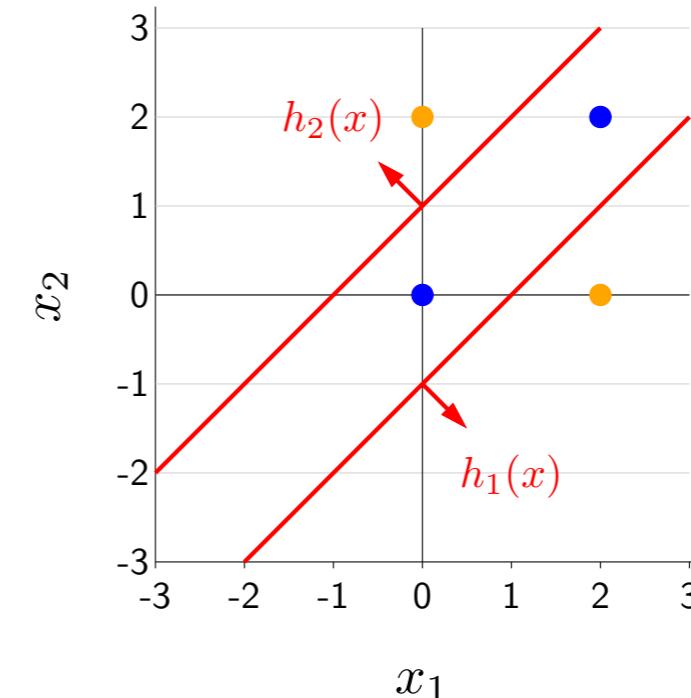
$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1]$$

Test if car 2 is far right of car 1:

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1]$$

Safe if at least one is true:

$$f(x) = \text{sign}(h_1(x) + h_2(x))$$



x	$h_1(x)$	$h_2(x)$	$f(x)$
$[0, 2]$	0	1	+1
$[2, 0]$	1	0	+1
$[0, 0]$	0	0	-1
$[2, 2]$	0	0	-1

- One way to motivate neural networks (without appealing to the brain) is **problem decomposition**.
- The intuition is to break up the full problem into two subproblems: the first subproblem tests if car 1 is to the far right of car 2; the second subproblem tests if car 2 is to the far right of car 1. Then the final output is 1 iff at least one of the two subproblems returns 1.
- Concretely, we can define $h_1(x)$ to be the output of the first subproblem, which is a simple linear decision boundary (in fact, the right line in the figure).
- Analogously, we define $h_2(x)$ to be the output of the second subproblem.
- Note that $h_1(x)$ and $h_2(x)$ take on values 0 or 1 instead of -1 or +1.
- The points can then be classified by first computing $h_1(x)$ and $h_2(x)$, and then combining the results into $f(x)$.

Rewriting using vector notation

Intermediate subproblems:

$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1] = \mathbf{1}[[\textcolor{red}{-1, +1, -1}] \cdot [1, x_1, x_2] \geq 0]$$

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1] = \mathbf{1}[[\textcolor{red}{-1, -1, +1}] \cdot [1, x_1, x_2] \geq 0]$$

$$\mathbf{h}(x) = \mathbf{1} \left[\begin{bmatrix} \textcolor{red}{-1} & \textcolor{red}{+1} & \textcolor{red}{-1} \\ \textcolor{red}{-1} & \textcolor{red}{-1} & \textcolor{red}{+1} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \geq 0 \right]$$

Predictor:

$$f(x) = \text{sign}(h_1(x) + h_2(x)) = \text{sign}([\textcolor{red}{1, 1}] \cdot \mathbf{h}(x))$$

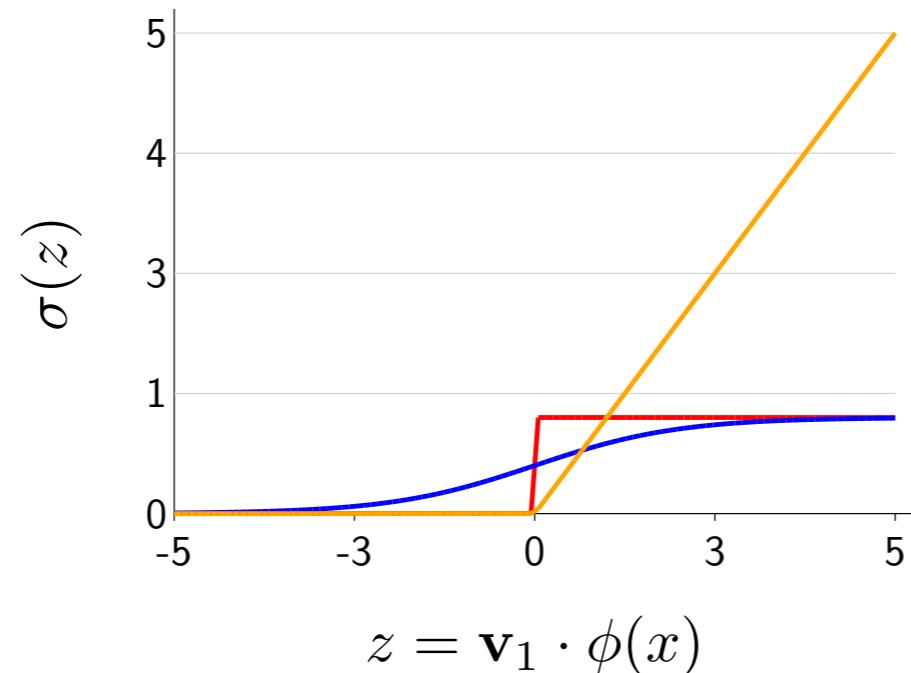
- Now let us rewrite this predictor $f(x)$ using vector notation.
- We can define a feature vector $[1, x_1, x_2]$ and a corresponding weight vector, where the dot product thresholded yields exactly $h_1(x)$.
- We do the same for $h_2(x)$.
- We put the two subproblems into one equation by stacking the weight vectors into one matrix. Recall that left-multiplication by a matrix is equivalent to taking the dot product with each row. By convention, the thresholding at 0 ($\mathbf{1}[\cdot \geq 0]$) applies component-wise.
- Finally, we can define the predictor in terms of a simple dot product.
- Now of course, we don't know the weight vectors, but we can learn them from the training data!

Avoid zero gradients

Problem: gradient of $h_1(x)$ with respect to \mathbf{v}_1 is 0

$$h_1(x) = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$

Solution: replace with an **activation function** σ with non-zero gradients



- Threshold: $\mathbf{1}[z \geq 0]$
- Logistic: $\frac{1}{1+e^{-z}}$
- ReLU: $\max(z, 0)$

$$h_1(x) = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

- Later we'll show how to perform learning using gradient descent, but we can anticipate one problem, which we encountered when we tried to optimize the zero-one loss.
- The gradient of $h_1(x)$ with respect to \mathbf{v}_1 is always zero because of the threshold function.
- To fix this, we replace the threshold function with an **activation function** with non-zero gradients
- Classically, neural networks used the **logistic function** $\sigma(z)$, which looks roughly like the threshold function but has non-zero gradients everywhere.
- Even though the gradients are non-zero, they can be quite small when $|z|$ is large (a phenomenon known as saturation). This makes optimizing with the logistic function still difficult.
- In 2012, Glorot et al. introduced the ReLU activation function, which is simply $\max(z, 0)$. This has the advantage that at least on the positive side, the gradient does not vanish (though on the negative side, the gradient is always zero). As a bonus, ReLU is easier to compute (only max, no exponentiation). In practice, ReLU works well and has become the activation function of choice.
- Note that if the activation function were linear (e.g., the identity function), then the gradients would always be nonzero, but you would lose the power of a neural network, because you would simply get the product of the final-layer weight vector and the weight matrix ($\mathbf{w}^T \mathbf{V}$), which is equivalent to optimizing over a single weight vector.
- Therefore, that there is a tension between wanting an activation function that is non-linear but also has non-zero gradients.

Two-layer neural networks

Intermediate subproblems:

$$\mathbf{h}(x) = \sigma(\mathbf{V} \phi(x))$$

The diagram illustrates the intermediate subproblem of a two-layer neural network. On the left, a vertical vector $\phi(x)$ is shown with three purple circles. In the center, a weight matrix \mathbf{V} is represented as a 3x5 grid of red circles. To the right, the resulting feature representation $\mathbf{h}(x)$ is shown as a vertical vector with five green circles.

Predictor (classification):

$$f_{\mathbf{V}, \mathbf{w}}(x) = \text{sign}(\mathbf{w}^\top \mathbf{h}(x))$$

The diagram shows the predictor function $f_{\mathbf{V}, \mathbf{w}}(x)$. It consists of a weight vector \mathbf{w} (represented by three red circles) multiplied by the feature representation $\mathbf{h}(x)$ (represented by three purple circles), followed by a sign function.

Interpret $\mathbf{h}(x)$ as a learned feature representation!

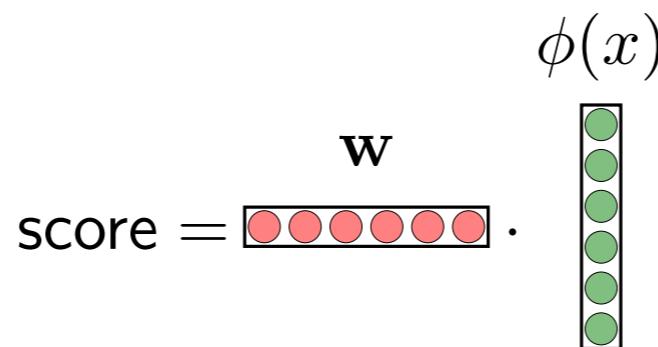
Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{V}, \mathbf{w}} : \mathbf{V} \in \mathbb{R}^{k \times d}, \mathbf{w} \in \mathbb{R}^k\}$$

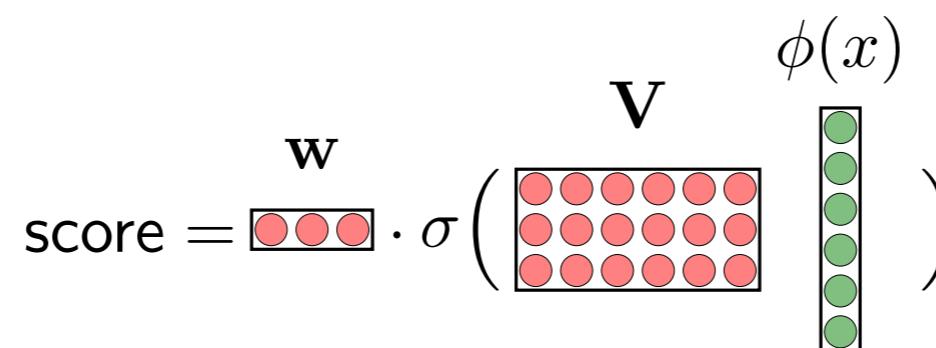
- Now we are finally ready to define the hypothesis class of two-layer neural networks.
- We start with a feature vector $\phi(x)$.
- We multiply it by a weight matrix \mathbf{V} (whose rows can be interpreted as the weight vectors of the k intermediate subproblems).
- Then we apply the activation function σ to each of the k components to get the hidden representation $\mathbf{h}(x) \in \mathbb{R}^k$.
- We can actually interpret $\mathbf{h}(x)$ as a learned feature vector (representation), which is derived from the original non-linear feature vector $\phi(x)$.
- Given $\mathbf{h}(x)$, we take the dot product with a weight vector \mathbf{w} to get the score used to drive either regression or classification.
- The hypothesis class is the set of all such predictors obtained by varying the first-layer weight matrix \mathbf{V} and the second-layer weight vector \mathbf{w} .

Deep neural networks

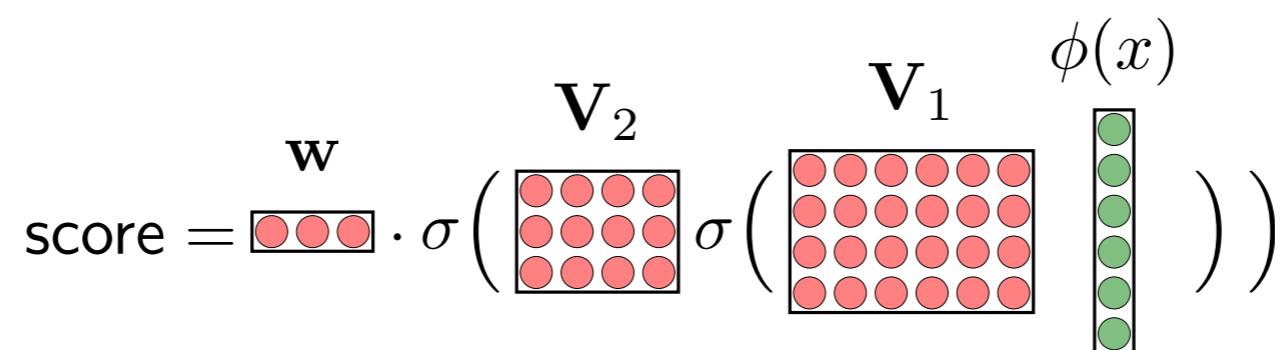
1-layer neural network:



2-layer neural network:

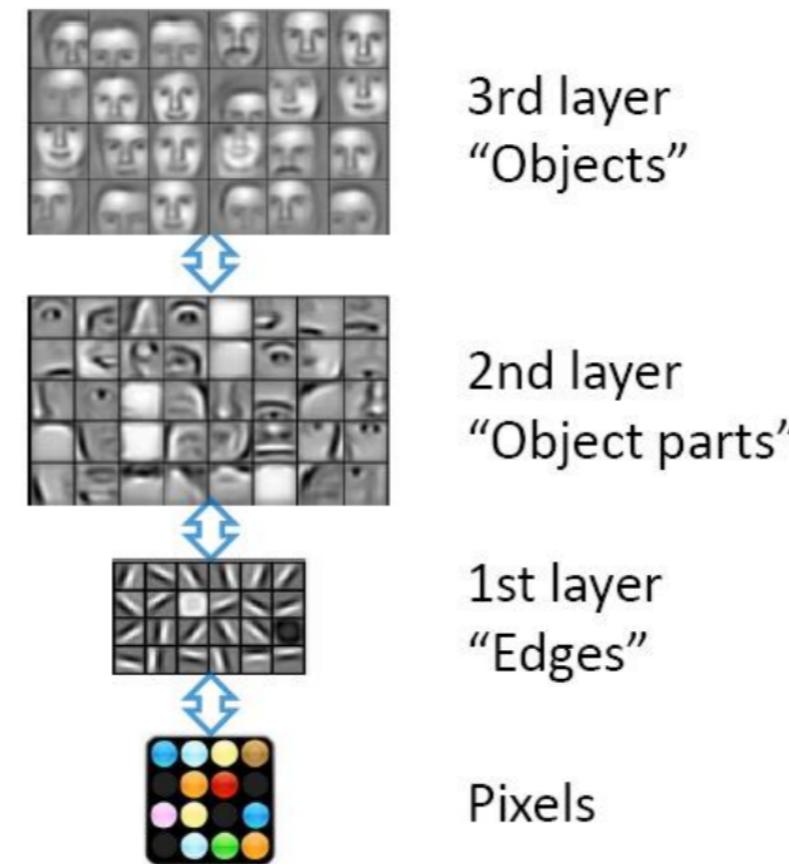


3-layer neural network:



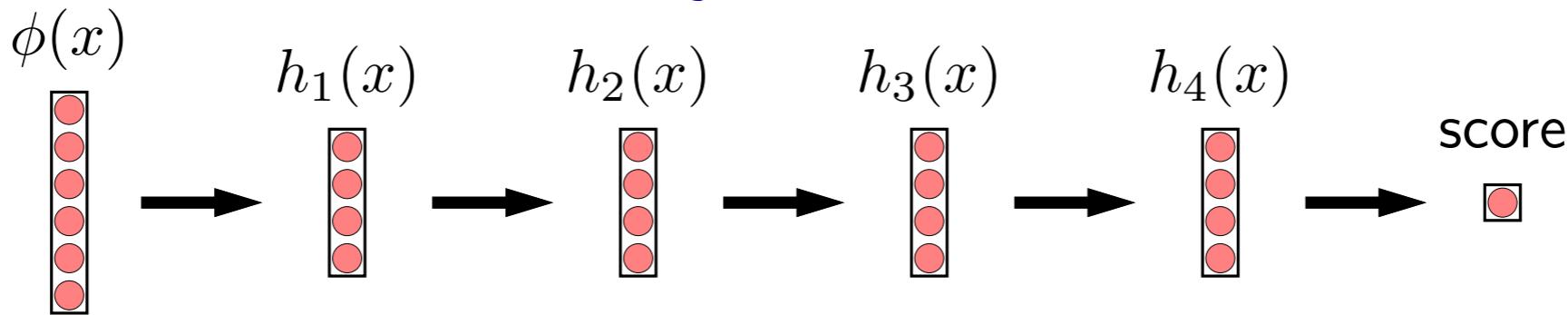
- We can push these ideas to build deep neural networks, which are neural networks with many layers.
- Warm up: for a one-layer neural network (a.k.a. a linear predictor), the score that drives prediction is simply a dot product between a weight vector and a feature vector.
- We just saw for a two-layer neural network, we apply a linear layer \mathbf{V} first, followed by a non-linearity σ , and then take the dot product.
- To obtain a three-layer neural network, we apply a linear layer and a non-linearity (this is the basic building block). This can be iterated any number of times. No matter how deep the neural network is, the top layer is always a linear function, and all the layers below that can be interpreted as defining a (possibly very complex) hidden feature vector.
- In practice, you would also have a bias term (e.g., $\mathbf{V}\phi(x) + b$). We have omitted all bias terms for notational simplicity.

Layers represent multiple levels of abstractions



- It can be difficult to understand what a sequence of (matrix multiply, non-linearity) operations buys you.
- To provide intuition, suppose the input feature vector $\phi(x)$ is a vector of all the pixels in an image.
- Then each layer can be thought of producing an increasingly abstract representation of the input. The first layer detects edges, the second detects object parts, the third detects objects. What is shown in the figure is for each component j of the hidden representation $\mathbf{h}(x)$, the input image $\phi(x)$ that maximizes the value of $h_j(x)$.
- Though we haven't talked about learning neural networks, it turns out that the "levels of abstraction" story is actually borne out visually when we learn neural networks on real data (e.g., images).

Why depth?



Intuitions:

- Multiple levels of abstraction
- Multiple steps of computation
- Empirically works well
- Theory is still incomplete

- Beyond learning hierarchical feature representations, deep neural networks can be interpreted in a few other ways.
- One perspective is that each layer can be thought of as performing some computation, and therefore deep neural networks can be thought of as performing multiple steps of computation.
- But ultimately, the real reason why deep neural networks are interesting is because they work well in practice.
- From a theoretical perspective, we have a quite an incomplete explanation for why depth is important. The original motivation from McCulloch/Pitts in 1943 showed that neural networks can be used to simulate a bounded computation logic circuit. Separately it has been shown that depth $k + 1$ logic circuits can represent more functions than depth k . However, neural networks are real-valued and might have types of computations which don't fit neatly into logical paradigm. Obtaining a better theoretical understanding is an active area of research in statistical learning theory.



Summary

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

The diagram illustrates the computation of a score. It shows a vector \mathbf{w} (represented by three red circles) being multiplied by the result of a sigmoid function σ . The argument of the sigmoid function is a matrix \mathbf{V} (represented by a 3x6 grid of red circles) multiplied by a feature vector $\phi(x)$ (represented by a vertical column of six green circles).

- Intuition: decompose problem into intermediate parallel subproblems
- Deep networks iterate this decomposition multiple times
- Hypothesis class contains predictors ranging over weights for all layers
- Next up: learning neural networks

- To summarize, we started with a toy problem (the XOR problem) and used it to motivate neural networks, which decompose a problem into intermediate subproblems, which are solved in parallel.
- Deep networks iterate this multiple times to build increasingly high-level representations of the input.
- Next, we will see how we can learn a neural network by choosing the weights for all the layers.



Machine learning: backpropagation



- In this module, I'll discuss **backpropagation**, an algorithm to automatically compute gradients.
- It is generally associated with training neural networks, but actually it is much more general and applies to any function.

Motivation: regression with four-layer neural networks

Loss on one example:

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x))))) - y)^2$$

Stochastic gradient descent:

$$\mathbf{V}_1 \leftarrow \mathbf{V}_1 - \eta \nabla_{\mathbf{V}_1} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_2 \leftarrow \mathbf{V}_2 - \eta \nabla_{\mathbf{V}_2} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_3 \leftarrow \mathbf{V}_3 - \eta \nabla_{\mathbf{V}_3} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

How to get the gradient without doing manual work?

- So far, we've defined neural networks, which take an initial feature vector $\phi(x)$ and sends it through a sequence of matrix multiplications and non-linear activations σ . At the end, we take the dot product between a weight vector \mathbf{w} to produce the score.
- In regression, we predict the score, and use the squared loss, which looks at the squared difference between the score and the target y .
- Recall that we can use stochastic gradient descent to optimize the training loss (which is an average over the per-example losses). Now, we need to update all the weight matrices, not just a single weight vector. This can be done by taking the gradient with respect to each weight vector/matrix separately, and updating the respective weight vector/matrix by subtracting the gradient times a step size η .
- We can now proceed to take the gradient of the loss function with respect to the various weight vector/matrices. You should know how to do this: just apply the chain rule. But grinding through this complex expression by hand can be quite tedious. If only we had a way for this to be done automatically for us...

Computation graphs

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x))))) - y)^2$$



Definition: computation graph

A directed acyclic graph whose root node represents the final mathematical expression and each node represents intermediate subexpressions.

Upshot: compute gradients via general **backpropagation** algorithm

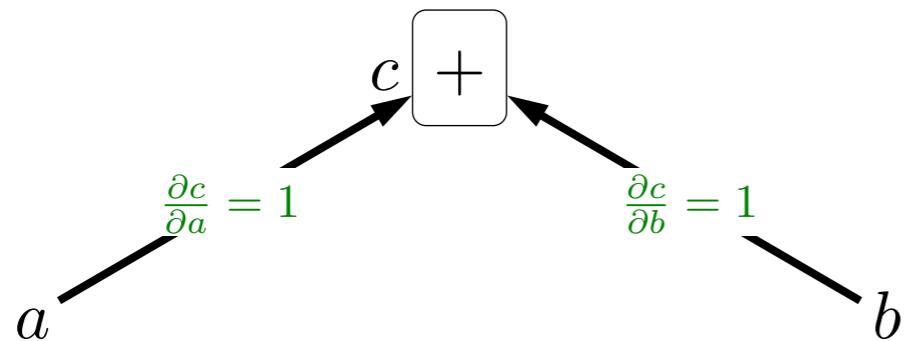
Purposes:

- Automatically compute gradients (how TensorFlow and PyTorch work)
- Gain insight into modular structure of gradient computations

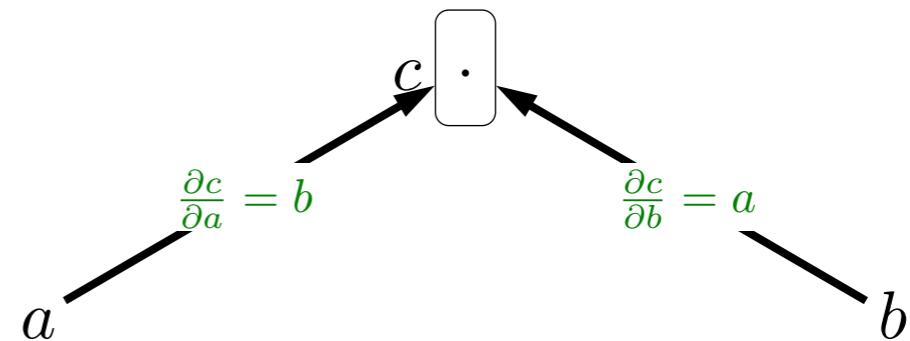
- Enter computation graphs, which will rescue us.
- A computation graph is a directed acyclic graph that represents an arbitrary mathematical expression. The root of that node represents the final expression, and the other nodes represent intermediate subexpressions.
- After having constructed the graph, we can compute all the gradients we want by running the general-purpose backpropagation algorithm, which operates on an arbitrary computation graph.
- There are two purposes to using computation graphs. The first and most obvious one is that it avoids having us to do pages of calculus, and instead delegates this to a computer. This is what packages such as TensorFlow or PyTorch do, and essentially all non-trivial deep learning models are trained like this.
- The second purpose is that by defining the graph, we can gain more insight into the nature of how gradients are computed in a modular way.

Functions as boxes

$$c = a + b$$



$$c = a \cdot b$$



$$(a + \epsilon) + b = c + 1\epsilon$$

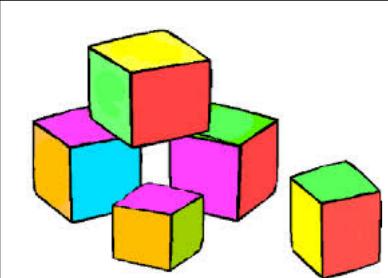
$$a + (b + \epsilon) = c + 1\epsilon$$

$$(a + \epsilon)b = c + b\epsilon$$

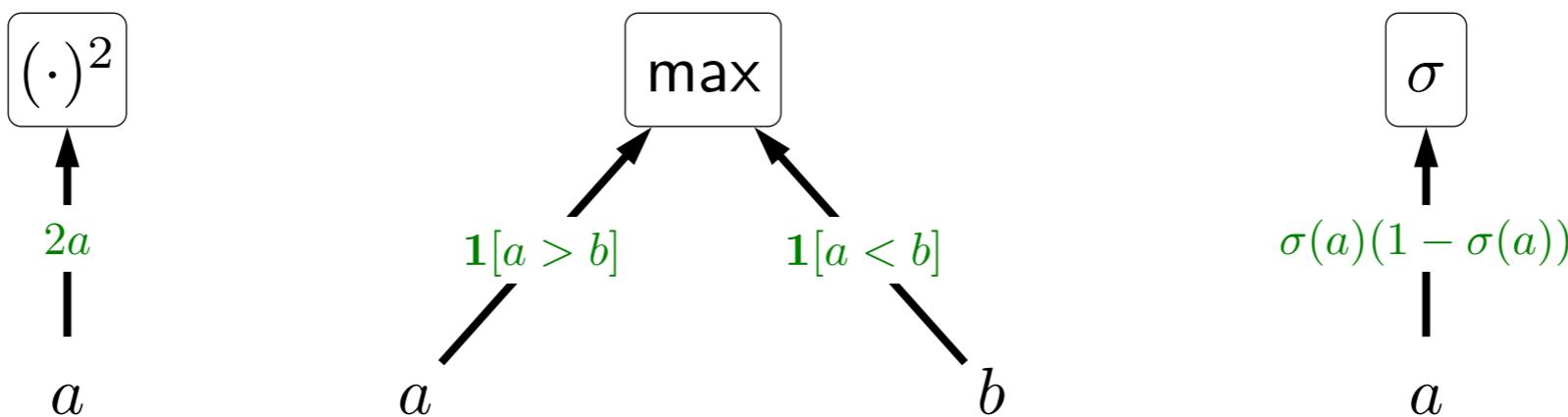
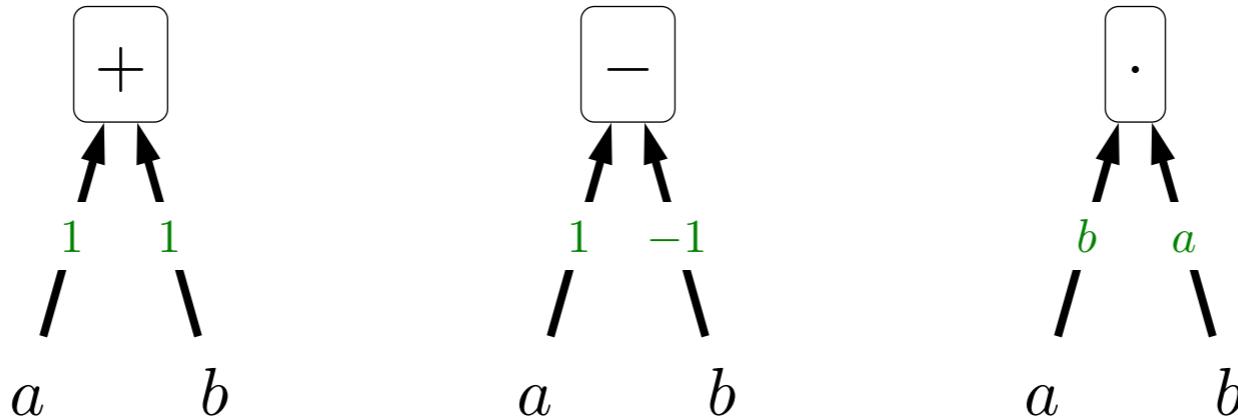
$$a(b + \epsilon) = c + a\epsilon$$

Gradients: how much does c change if a or b changes?

- The first conceptual step is to think of functions as boxes that take a set of inputs and produces an output.
- For example, take $c = a + b$. The key question is: if we perturb a by a small amount ϵ , how much does the output c change? In this case, the output c is also perturbed by 1ϵ , so the gradient (partial derivative) is 1. We put this gradient on the edge.
- We can handle $c = a \cdot b$ in a similar way.
- Intuitively, the gradient is a measure of local sensitivity: how much input perturbations get amplified when they go through the various functions.



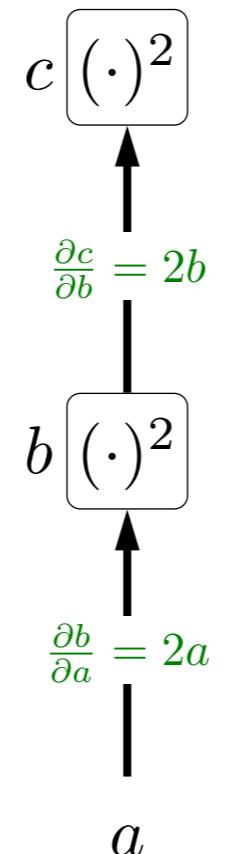
Basic building blocks



- Here are some more examples of simple functions and their gradients. Let's walk through them together.
- These should be familiar from basic calculus. All we've done is present them in a visually more intuitive way.
- For the max function, changing a only impacts the max iff $a > b$; and analogously for b .
- For the logistic function $\sigma(z) = \frac{1}{1+e^{-z}}$, a bit of algebraic elbow grease produces the gradient. You can check that the gradient is zero when $|a| \rightarrow \infty$.
- It turns out that these simple functions are all we need to build up many of the more complex and potentially scarier looking functions that we'll encounter.



Function composition

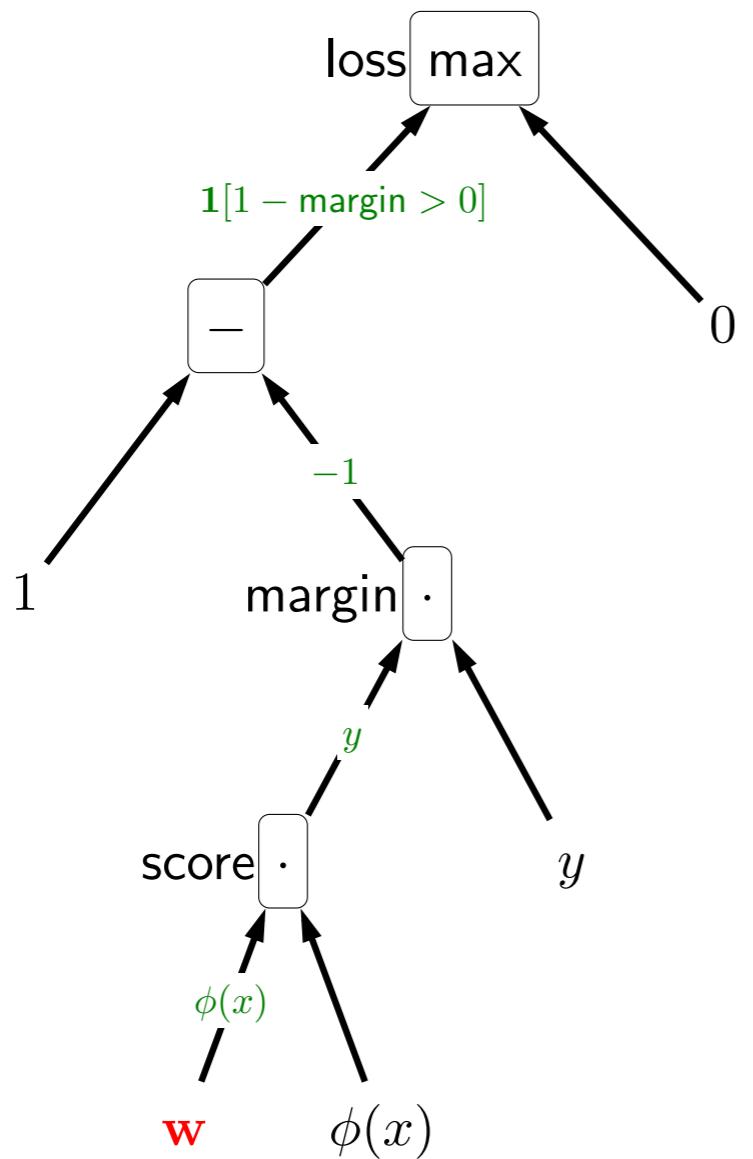


Chain rule:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} = (2b)(2a) = 4a^3$$

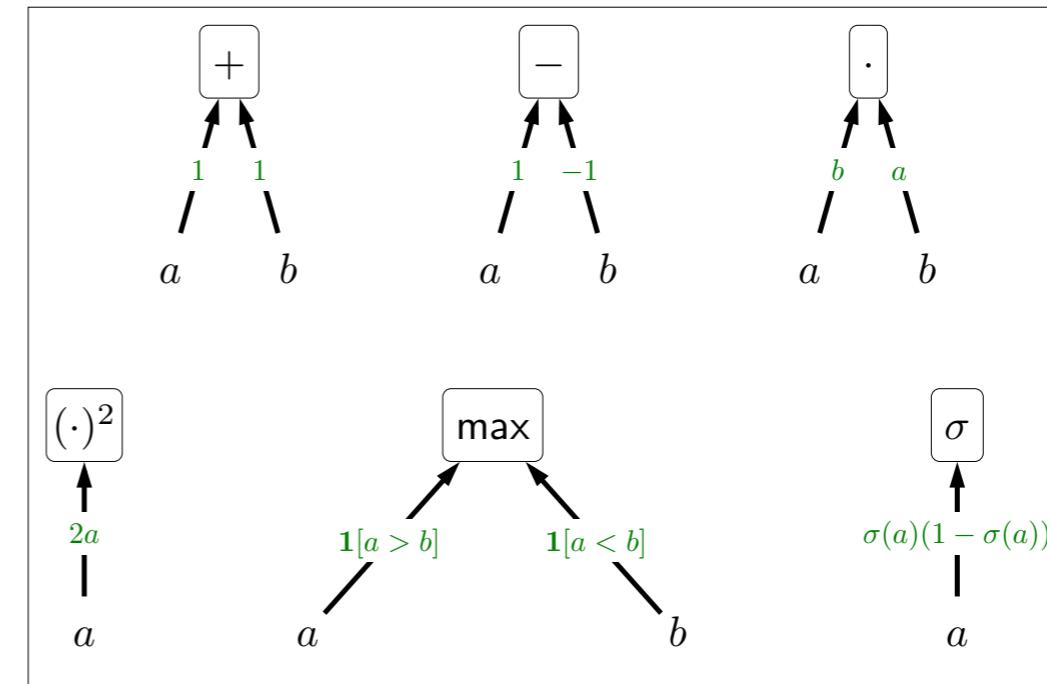
- Given these building blocks, we can now put them together to create more complex functions.
- Consider applying some function (e.g., squared) to a to get b , and then applying some other function (e.g., squared) to get c .
- What is the gradient of c with respect to a ?
- We know from our building blocks the gradients on the edges.
- The final answer is given by the **chain rule** from calculus: just multiply the two gradients together.
- You can verify that this yields the correct answer $(2b)(2a) = 4a^3$.
- This visual intuition will help us better understand more complex functions.

Linear classification with hinge loss



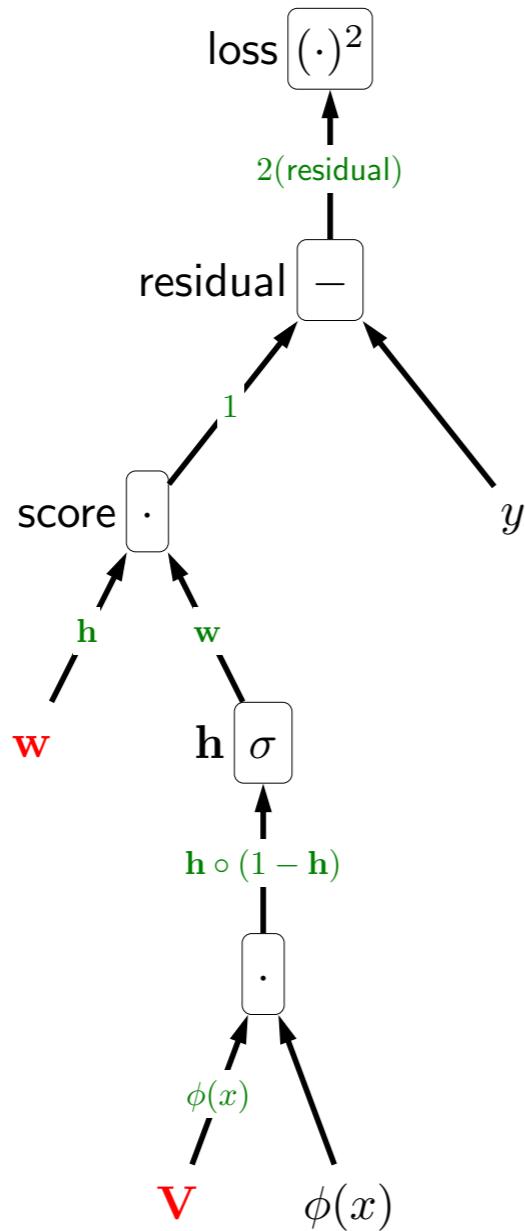
$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x)y, 0\}$$

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = -\mathbf{1}[margin < 1]\phi(x)y$$



- Now let's turn to our first real-world example: the hinge loss for linear classification. We already computed the gradient before, but let's do it using computation graphs.
- We can construct the computation graph for this expression, proceeding bottom up. At the leaves are the inputs and the constants. Each internal node is labeled with the operation (e.g., \cdot) and is labeled with a variable naming that subexpression (e.g., margin).
- In red, we have highlighted the weights w with respect to which we want to take the gradient. The central question is how small perturbations in w affect a change in the output (loss).
- We can examine each edge from the path from w to loss, and compute the gradient using our handy reference of building blocks.
- The actual gradient is the product of the edge-wise gradients from w to the loss output.

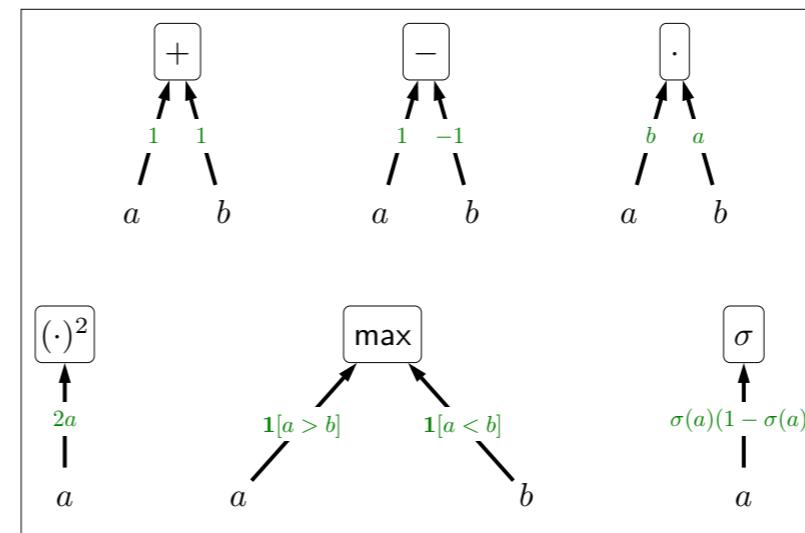
Two-layer neural networks



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

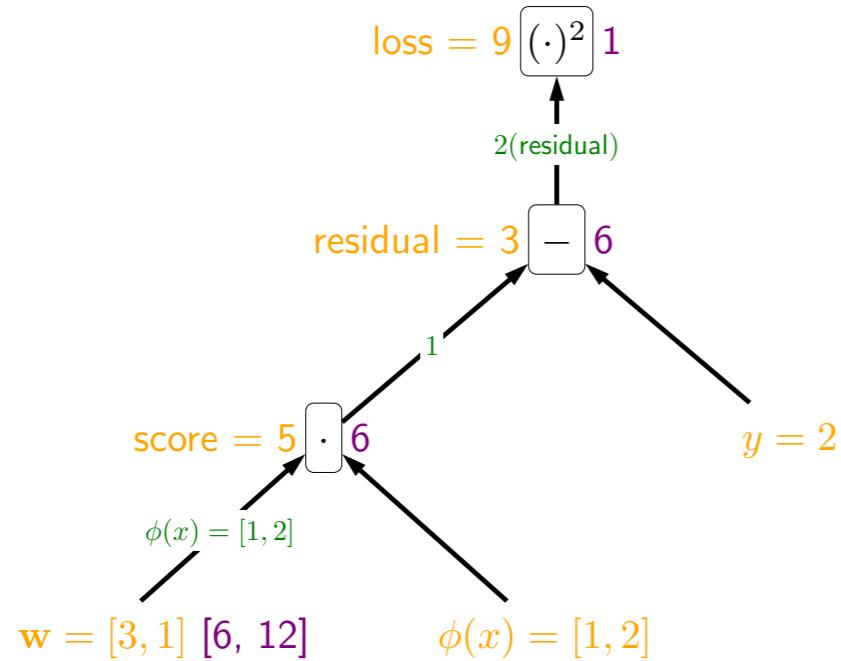
$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{h}$$

$$\nabla_{\mathbf{V}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{w} \circ \mathbf{h} \circ (1 - \mathbf{h})\phi(x)^\top$$



- We now finally turn to neural networks, but the idea is essentially the same.
- Specifically, consider a two-layer neural network driving the squared loss.
- Let us build the computation graph bottom up.
- Now we need to take the gradient with respect to \mathbf{w} and \mathbf{V} . Again, these are just the product of the gradients on the paths from \mathbf{w} or \mathbf{V} to the loss node at the root.
- Note that the two gradients have in common the first two terms. Common paths result in common subexpressions for the gradient.
- There are some technicalities when dealing with vectors worth mentioning: First, the \circ in $\mathbf{h} \circ (1 - \mathbf{h})$ is elementwise multiplication (not the dot product), since the non-linearity σ is applied elementwise. Second, there is a transpose for the gradient expression with respect to \mathbf{V} and not \mathbf{w} because we are taking $\mathbf{V}\phi(\mathbf{x})$, while taking $\mathbf{w} \cdot \mathbf{h} = \mathbf{w}^\top \mathbf{h}$.
- This computation graph also highlights the modularity of hypothesis class and loss function. You can pick any hypothesis class (linear predictors or neural networks) to drive the score, and the score can be fed into any loss function (squared, hinge, etc.).

Backpropagation

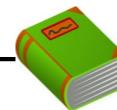


$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$$

↓
backpropagation

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$$



Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{loss}}{\partial f_i}$ is how f_i influences loss



Algorithm: backpropagation algorithm

Forward pass: compute each f_i (from leaves to root)

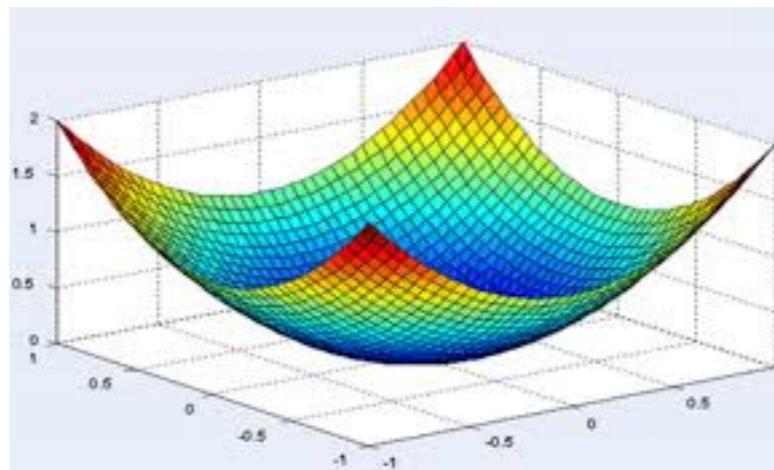
Backward pass: compute each g_i (from root to leaves)

- So far, we have mainly used the graphical representation to visualize the computation of function values and gradients for our conceptual understanding.
- Now let us introduce the **backpropagation** algorithm, a general procedure for computing gradients given only the specification of the function.
- Let us go back to the simplest example: linear regression with the squared loss.
- All the quantities that we've been computing have been so far symbolic, but the actual algorithm works on real numbers and vectors. So let's use concrete values to illustrate the backpropagation algorithm.
- The backpropagation algorithm has two phases: forward and backward. In the forward phase, we compute a **forward value** f_i for each node, corresponding to the evaluation of that subexpression. Let's work through the example.
- In the backward phase, we compute a **backward value** g_i for each node. This value is the gradient of the loss with respect to that node, which is also the product of all the gradients on the edges from the node to the root. To compute this backward value, we simply take the parent's backward value and multiply by the gradient on the edge to the parent. Let's work through the example.
- Note that both f_i and g_i can either be scalars, vectors, or matrices, but have the same dimensionality.

A note on optimization

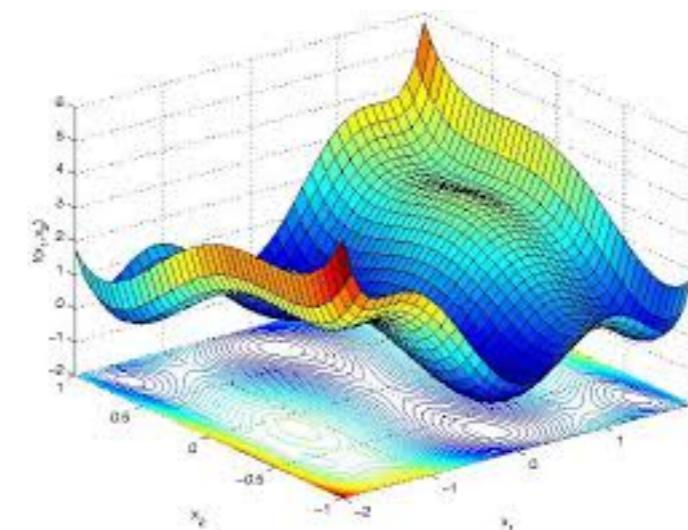
$$\min_{\mathbf{V}, \mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

Linear predictors



(convex)

Neural networks



(non-convex)

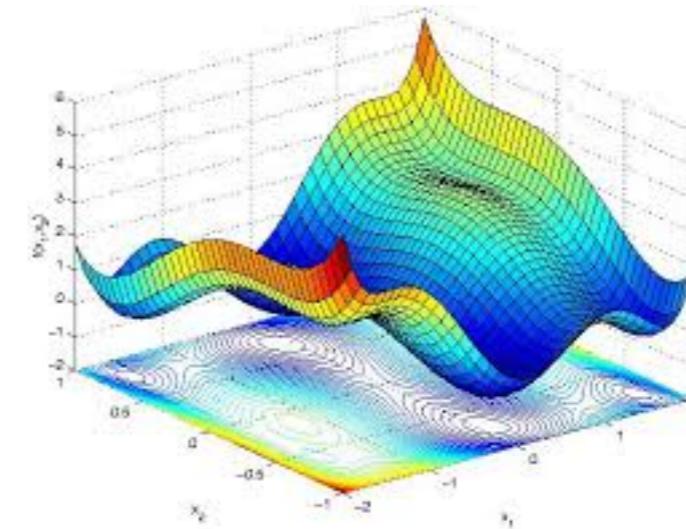
Optimization of neural networks is in principle hard

- So now we can apply the backpropagation algorithm and compute gradients, stick them into stochastic gradient descent, and get some answer out.
- One question which we haven't addressed is whether stochastic gradient descent will work in the sense of actually finding the weights that minimize the training loss?
- For linear predictors (using the squared loss or hinge loss), $\text{TrainLoss}(\mathbf{w})$ is a convex function, which means that SGD (with an appropriately step size) is theoretically guaranteed to converge to the global optimum.
- However, for neural networks, $\text{TrainLoss}(\mathbf{V}, \mathbf{w})$ is typically non-convex which means that there are multiple local optima, and SGD is not guaranteed to converge to the global optimum. There are many settings that SGD fails both theoretically and empirically, but in practice, SGD on neural networks can work much better than theory would predict, provided certain precautions are taken. The gap between theory and practice is not well understood and an active area of research.

How to train neural networks

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

The diagram illustrates the computation of a neural network score. It shows a weight vector \mathbf{w} (represented by three red circles) being multiplied by the output of a hidden layer $\phi(x)$ (represented by a vertical stack of green circles). The hidden layer $\phi(x)$ is produced by applying an activation function σ to the input x , which is represented by a matrix \mathbf{V} containing red circles.



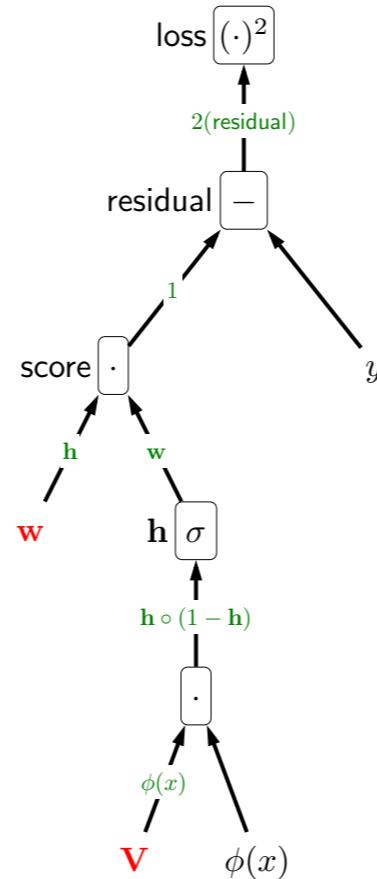
- Careful initialization (random noise, pre-training)
- Overparameterization (more hidden units than needed)
- Adaptive step sizes (AdaGrad, Adam)

Don't let gradients vanish or explode!

- Training a neural network is very much like driving stick. In practice, there are some "tricks" that are needed to make things work properly. Just to name a few to give you a sense of the considerations:
- Initialization (where you start the weights) matters for non-convex optimization. Unlike for linear models, you can't start at zero or else all the subproblems will be the same (all rows of \mathbf{V} will be the same). Instead, you want to initialize with a small amount of random noise.
- It is common to use overparameterized neural networks, ones with more hidden units (k) than is needed, because then there are more "chances" that some of them will pick out on the right signal, and it is okay if some of the hidden units become "dead".
- There are small but important extensions of stochastic gradient descent that allow the step size to be tuned per weight.
- Perhaps one high-level piece of advice is that when training a neural network, it is important to monitor the gradients. If they vanish (get too small), then training won't make progress. If they explode (get too big), then training will be unstable.

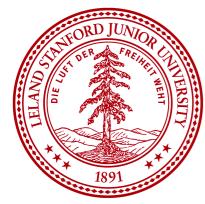


Summary



- Computation graphs: visualize and understand gradients
- Backpropagation: general-purpose algorithm for computing gradients

- The most important concept in this module is the idea of a **computation graph**, allows us to represent arbitrary mathematical expressions, which can just be built out of simple building blocks. They hopefully have given you a more visual and better understanding of what gradients are about.
- The **backpropagation** algorithm allows us to simply write down an expression, and never have to take a gradient manually again. However, it is still important to understand how the gradient arises, so that when you try to train a deep neural network and your gradients vanish, you know how to think about debugging your network.
- The generality of computation graphs and backpropagation makes it possible to iterate very quickly on new types of models and loss functions and opens up a new paradigm for model development: **differential programming**.

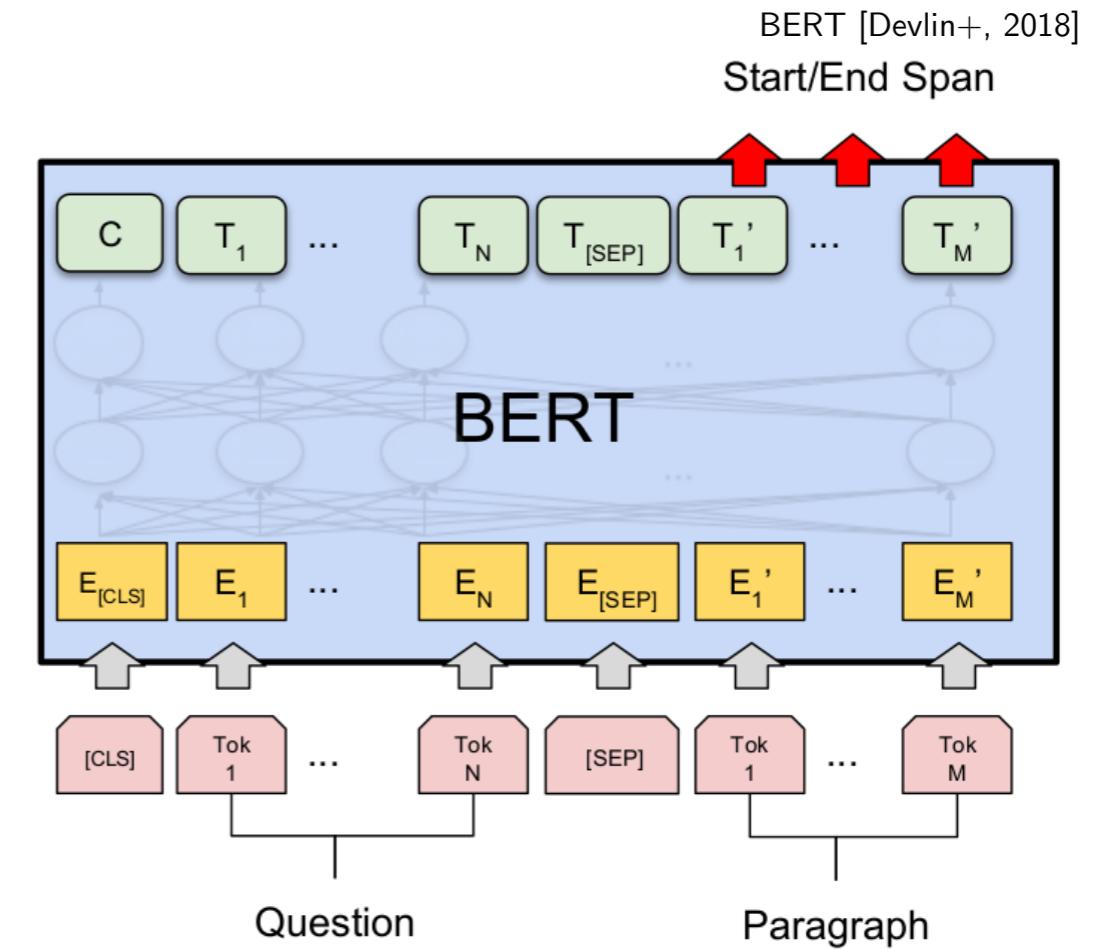
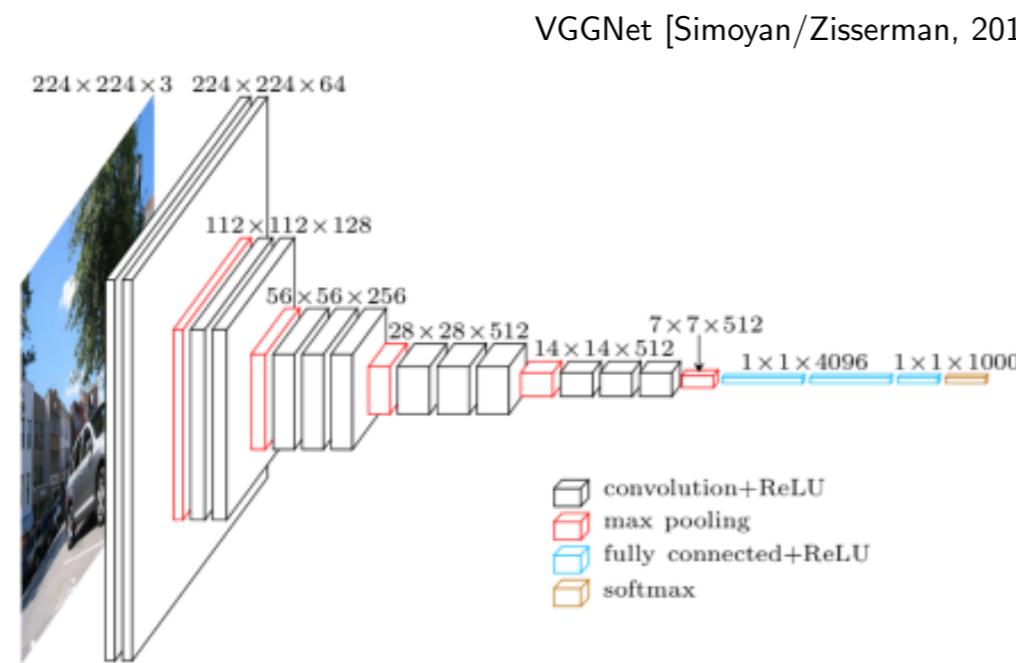


Machine learning: differentiable programming



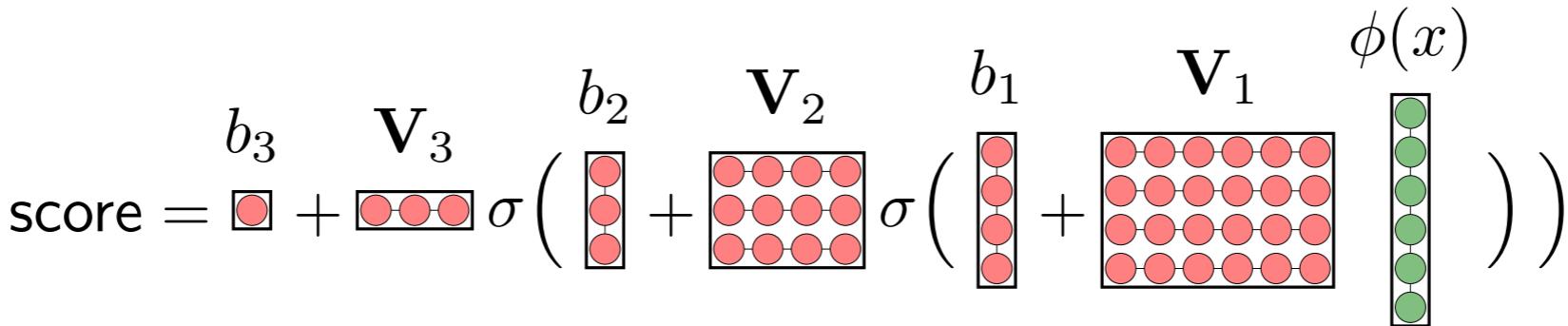
- In this module, I'll briefly introduce the idea of differentiable programming, which runs with the ideas of computation graphs and backpropagation that we developed for simple neural networks.
- There is enough to say here to fill up an entire course, so I will keep things high-level but try to highlight the power of **composition**.
- Aside: Differentiable programming is closely related to deep learning. I've adopted the former term as a more precise way to highlight the mechanics of writing models as you would write code.

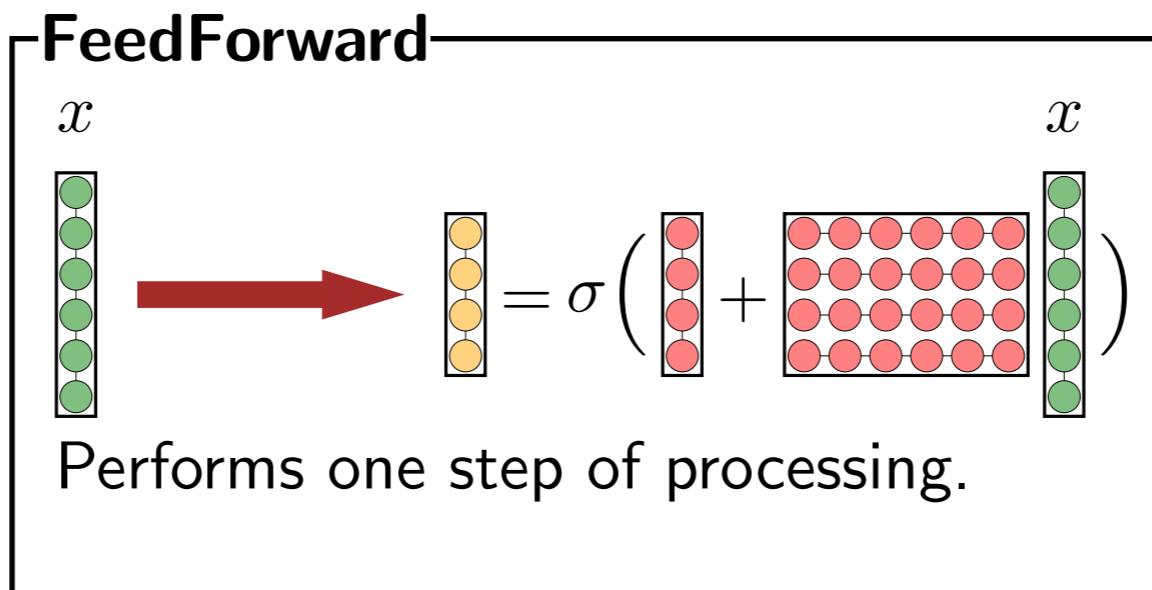
Deep learning models



- If you look around at deep learning today, there are some pretty complex models which have many layers, attention mechanisms, residual connections, layerwise normalization, to name a few, which might be overwhelming at first glance.
- However, if you look closer, these complex models are actually composed of functions, which themselves are composed of even simpler functions.
- This is the "programming" part of differentiable programming, which allows you to build up increasingly more powerful and sophisticated models without losing track of what's going on.

Feedforward neural networks

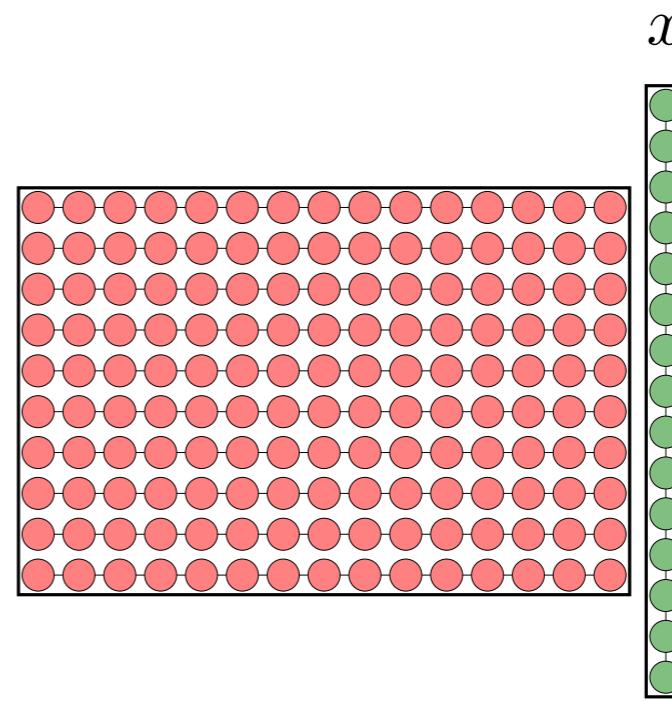
$$\text{score} = b_3 + \mathbf{V}_3 \sigma \left(b_2 + \mathbf{V}_2 \sigma \left(b_1 + \mathbf{V}_1 \phi(x) \right) \right)$$




$$\text{score} = \text{FeedForward}(\text{FeedForward}(\text{FeedForward}(\phi(x)))) = \text{FeedForward}^3(\phi(x))$$

- Let's revisit our familiar example, the three-layer neural network. In this model, we start with the feature vector $\phi(x)$ apply some operations (left-multiply by a matrix, add a bias term) to get the score.
- (Note that we highlight that each of these matrices should be interpreted as a collection of rows.)
- The main differences from before are that we've added bias terms and renamed the final weight vector w to V_3 for consistency with the other layers.
- Let us now factor out a function of this model and call it **FeedForward**. This is a function that takes a fixed-dimensional vector x and produces another vector (with potentially another dimensionality). This function is implemented by multiplying by a matrix and applying an activation function (e.g., ReLU).
- Now we can not worry too much about the implementation, and just think of this as performing one step of processing. Compared to normal programming, this is admittedly vague, because the parameters (e.g., the red quantities) are not specified but rather learned from data. So differentiable programming requires us to necessarily be a bit loose.
- Using this brand new function, we can rewrite the three-layer neural network as follows. Strictly speaking, when we write **FeedForward**, we mean a function that has its own private parameters that need to be set via backpropagation.
- Another simplification is that we are eliding the input and output dimensionality (6 to 4, 4 to 3, and 3 to 1 in this example). These are hyperparameters which need to be set before learning.

Representing images

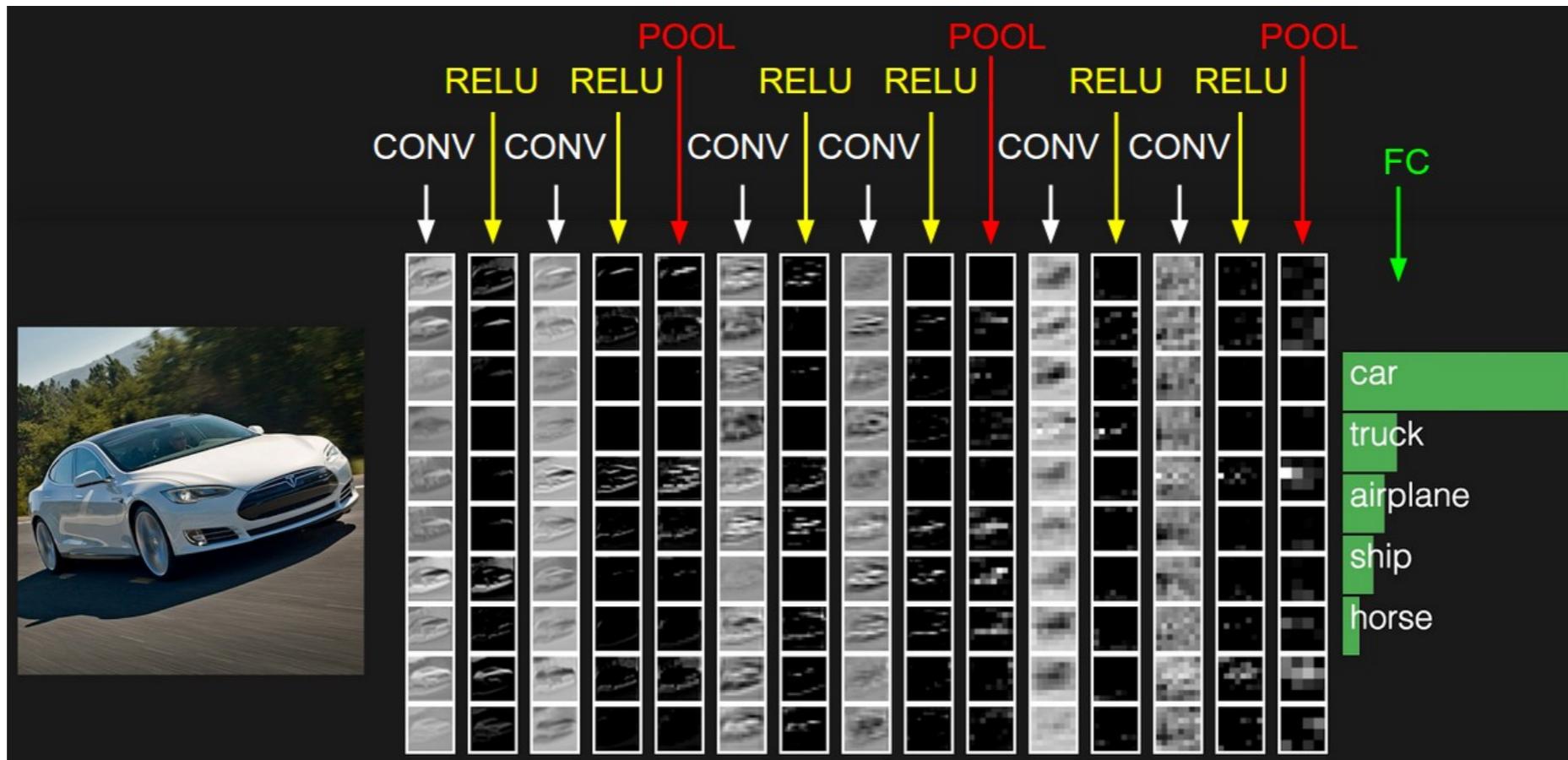


Problems:

- Matrix is huge (depending on resolution of image)
- Does not capture the spatial structure (locality) of images

- Now suppose we want to do image classification. We need to have some way of representing images.
- The **FeedForward** function takes in a vector as input, and we can represent an image as a long vector containing all the pixels (say, by concatenating all the rows).
- But then we would then have to have a huge matrix to transform this input, resulting in a lot of parameters (especially if the image is high resolution), which might be difficult to learn.
- The issue is that we're not leverage knowledge that these are images. If you permute the components of the input vector x and re-train, you will just get a permuted parameters, but the same predictions.

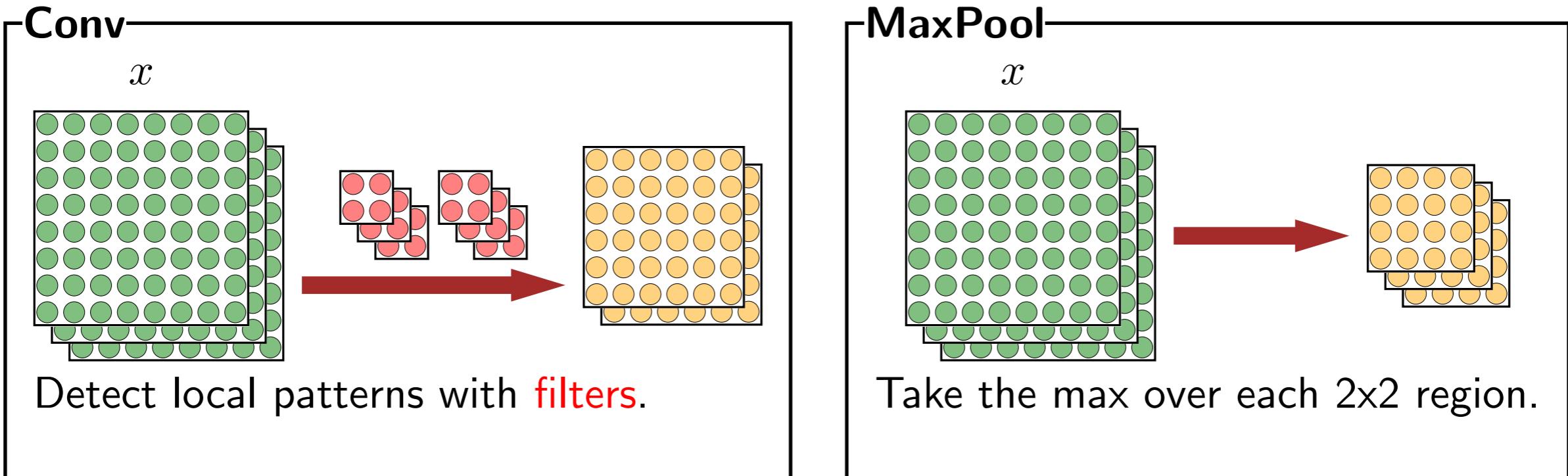
Convolutional neural networks



[Andrej Karpathy's demo]

- Convolutional neural networks (ConvNets or CNNs) is a refinement of the vanilla fully-connected neural networks tailored for images.
- It is also used for sequences such as text (everything is just 1D instead of 2D) or video (everything is 3D instead of 2D).
- Here is a visualization of a ConvNet making a prediction on this car image. There are a sequence of layers which turn the image into something increasingly abstract, and finally we get a vector representing the probabilities of the different object categories.
- You can click on this link to see Andrej Karpathy's demo, where you can create and train ConvNets in your browser.

Convolutional neural networks



[Andrej Karpathy's demo]

$$\text{AlexNet}(x) = \text{FeedForward}^3(\text{MaxPool}(\text{Conv}^3(\text{MaxPool}(\text{Conv}(\text{MaxPool}(\text{Conv}(x)))))))$$

- So let us now define the two basic building blocks of ConvNets. We're not going to go through the details, but simply focus on the interface. (You should take CS231N if you want to learn more about ConvNets.)
- First, **Conv** takes as input an image, which can be represented as a **volume**, which is a matrix for each channel (red, green, blue), and each matrix has same height and width as the image.
- This function produces another volume whose height and width are either the same or a bit smaller than that of the input volume, and the number of output channels could be different.
- The output volume is constructed by sweeping a **filter** over the input volume, and taking the dot product of the filter with a local part of the input volume. That produces a number which you write into the output volume. The number of filters determines the number of channels in the output volume.
- The second operation is **MaxPool**, which simply takes an input volume and reduces the width and height by taking the max over local regions.
- Note that MaxPool does not have any parameters and has the same number of input channels as output channels.
- Given just these two functions, **Conv** and **MaxPool**, as well as our friend **FeedForward**, we can express the famous AlexNet architecture, which won the ImageNet competition in 2012 and arguably kicked off deep learning revolution.
- Note that each of these functions has its own parameters that need to be trained.
- Each one also has its own hyperparameters (number of channels, filter size, etc.).

Representing natural language

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?

gravity

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?

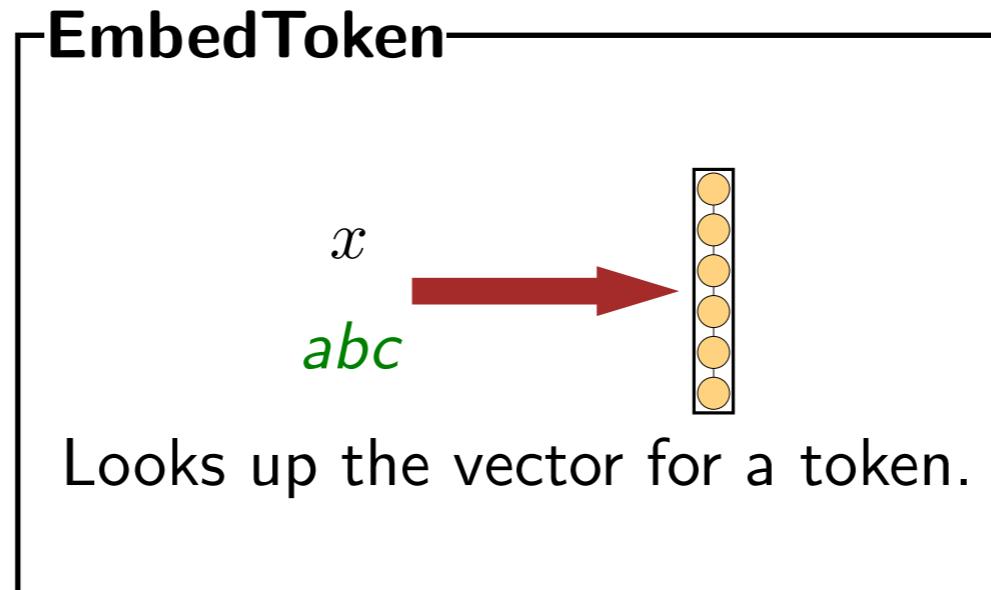
graupel

Where do water droplets collide with ice crystals to form precipitation?

within a cloud

- Now let us turn our attention to natural language processing. As a motivating example, suppose we wanted to build a question answering system.
- Here are examples of questions from the SQuAD question answering benchmark, where the input is the paragraph, a question, and the goal is to select the span of the paragraph that answers the question.
- This requires somehow relating the paragraph to the question; sometimes string match will work (e.g., "precipitation"), and sometimes they don't (e.g., "causes" and "product").

Embedding tokens



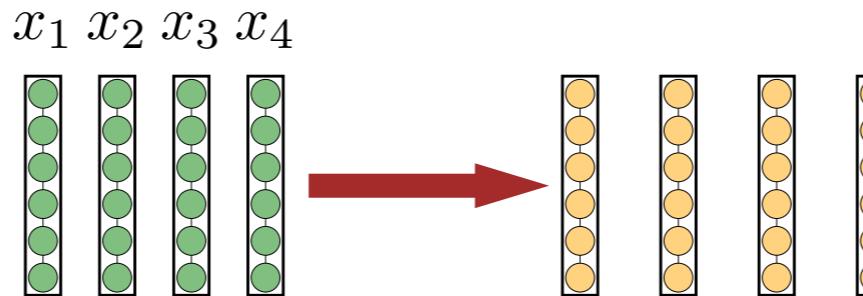
In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

Meaning of words/tokens depends on context...

- Words are discrete objects, whereas neural networks speak vectors.
- So the first step is to embed the words (more generally, tokens) into vectors. This is usually just accomplished by looking up the token in a dictionary that maps tokens to vectors.
- Now given a sentence (sequence of tokens), we just embed each of the tokens independently to produce a sequence of vectors.
- However, this is not quite satisfactory because the meaning of words depends on context. For example, "product" could mean result or it could mean multiplication.

Representing sequences

SequenceModel



Process each element of a sequence with respect to other elements.

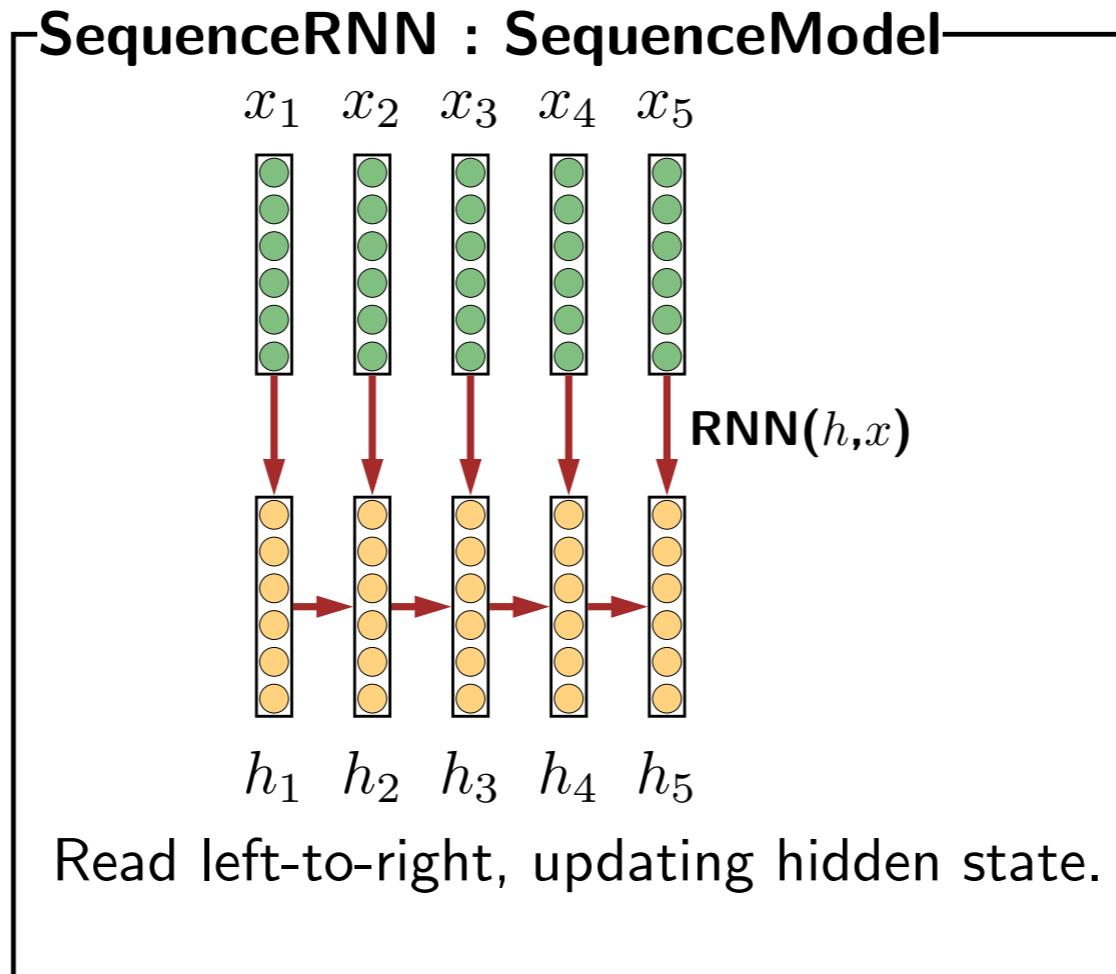
Two implementations:

- Recurrent neural networks
- Transformers

- We're now going to define an abstract function **SequenceModel** (to continue the programming metaphor).
- **SequenceModel** takes a sequence of vectors and produces a corresponding sequence of vectors, where each vector has been "contextualized" with respect to the other vectors.
- We will see two implementations, recurrent neural networks and Transformers.

Recurrent neural networks

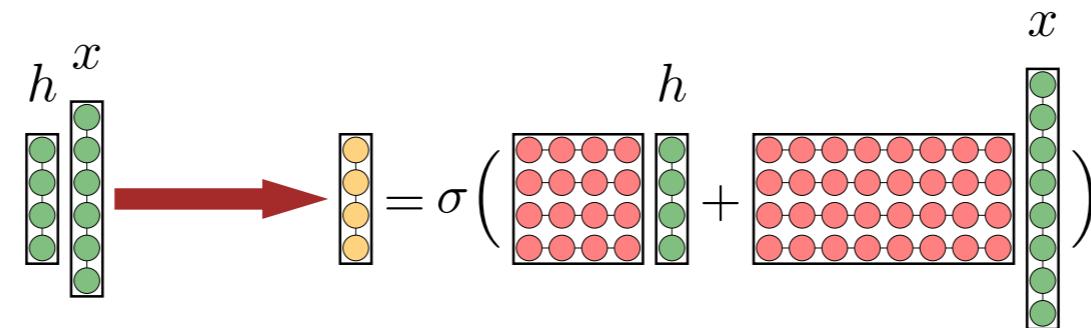
In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.



- A recurrent neural network (RNN) can be thought of as reading left to right. It maintains a hidden state which represents the past and gets updated with each new input vector.
- At the end of the day, however, it still produces a sequence of (contextualized) vectors given a sequence of input vectors.
- We still have to specify how the RNN computes the new hidden state given the old hidden state and the new input.

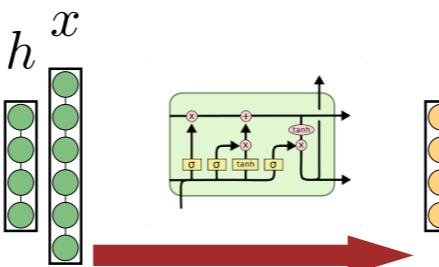
Recurrent neural networks

SimpleRNN : RNN



Update hidden state given a new input.

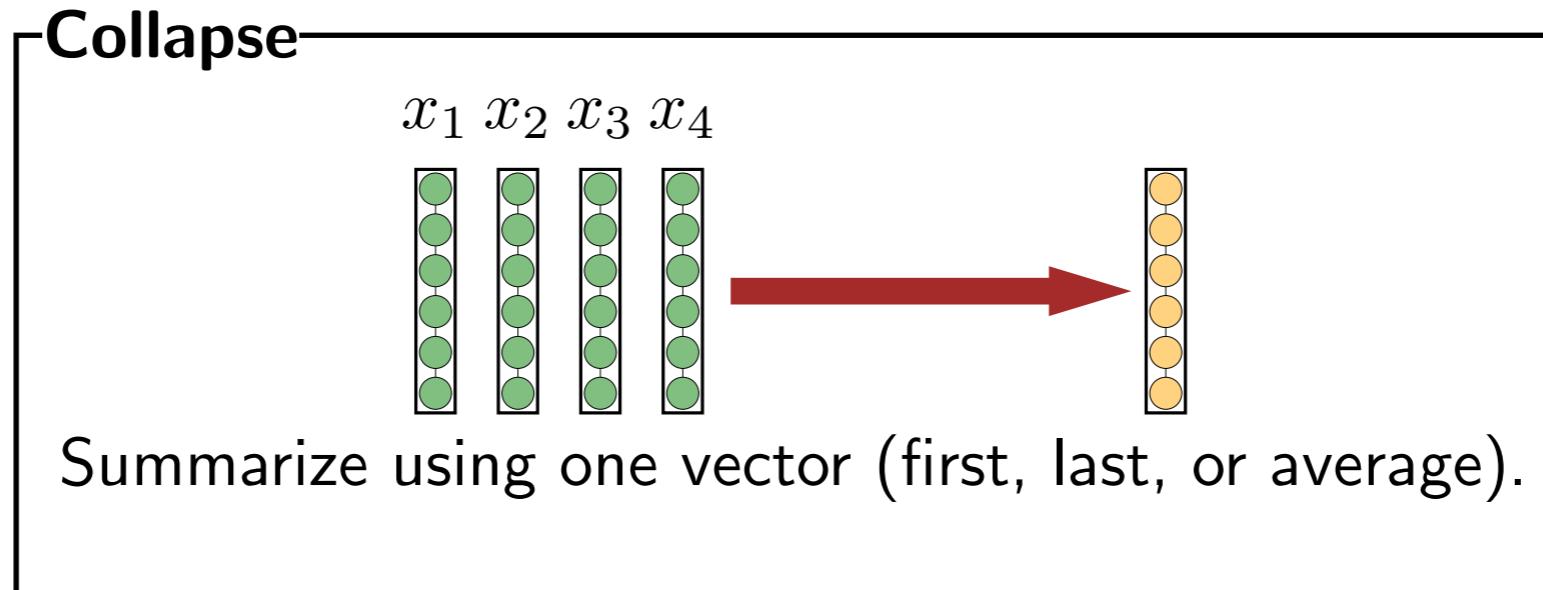
LSTM : RNN



Update hidden state given a new input without forgetting the past.

- There are two types of RNNs I will talk about.
- A Simple RNN takes the old hidden state h , a new input x and produces a new hidden state.
- Both h and x get multiplied by a vector; the result is summed, and we apply the activation function as usual.
- Note that this is equivalent to just applying **FeedForward** on the concatenation of h and x .
- One problem with simple RNNs is that they suffer from the vanishing gradient problem, which makes them hard to train.
- Long Short-Term Memory (LSTMs) were developed in response to this problem. We won't go over the details, but will just say that it privileges h and makes sure that h doesn't forget the history of inputs.

Collapsing to a single vector



Example text classification model:

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

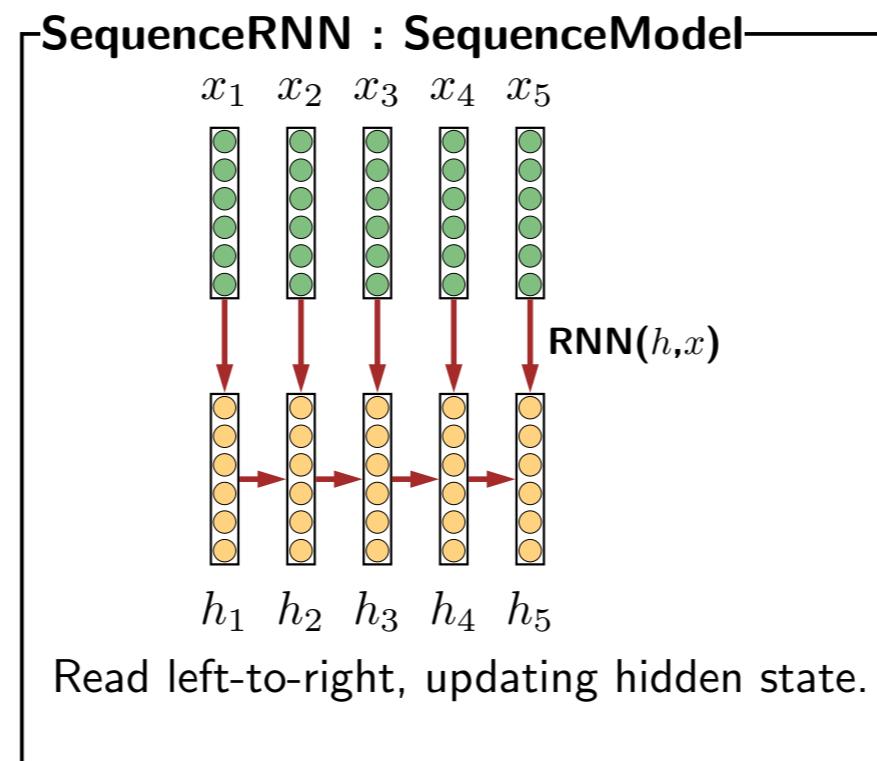
$$\text{score} = \mathbf{w} \cdot \mathbf{Collapse}(\mathbf{SequenceModel}^3(\mathbf{EmbedToken}(\mathbf{x})))$$

- Note that the sequence model produces a variable number of vectors. In order to do classification, we need to **Collapse** the vectors into one, which can then be used to drive the score for classification.
- There are three common things that are done: return the first vector, return the last vector, or return the average of all of them.

Long-range dependencies

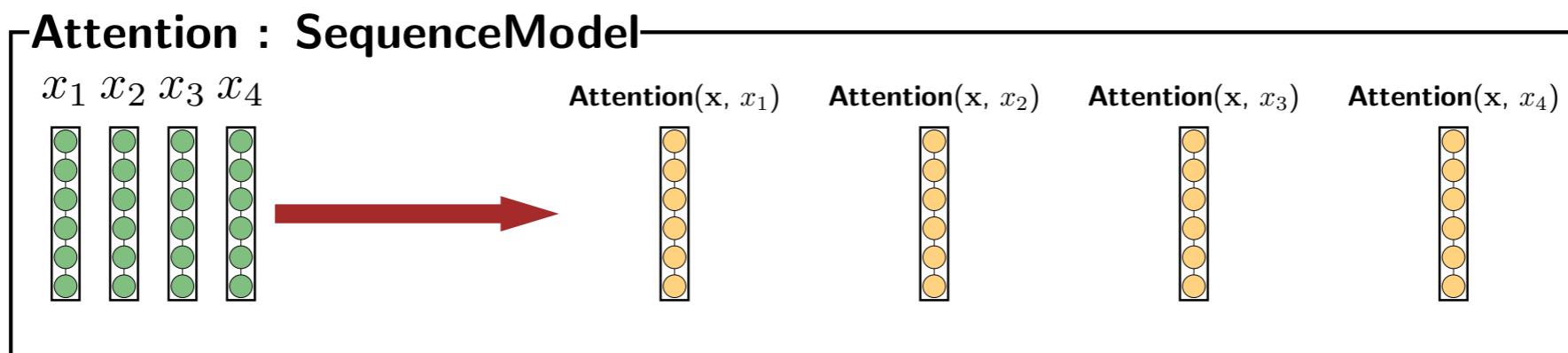
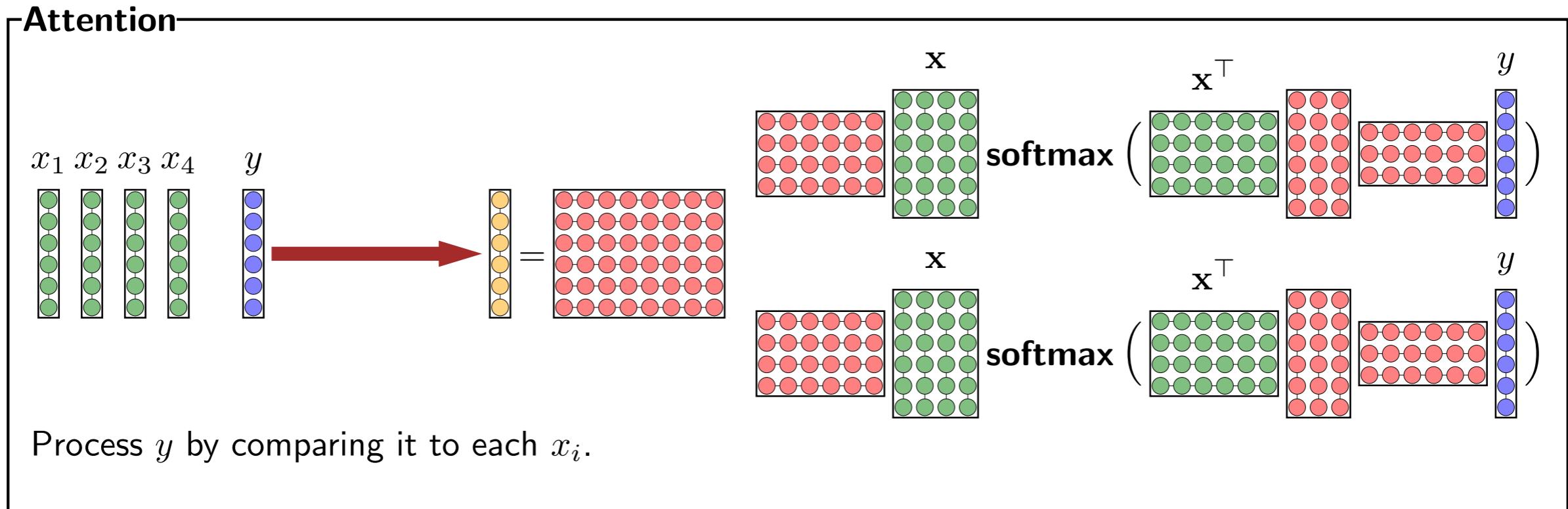
[CLS] What causes precipitation to fall? [SEP] In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

Problem: RNN (and ConvNets) are very local



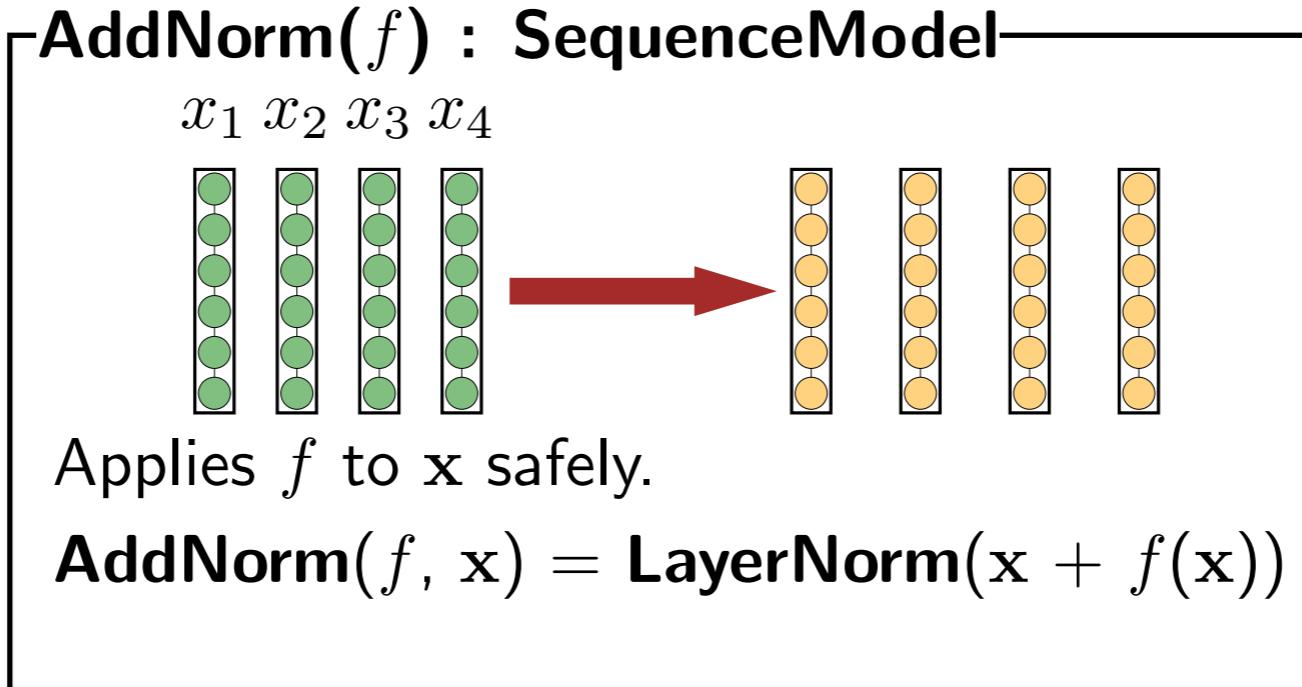
- This is all nice, but there is one big problem with RNNs, which is that they are very local, meaning that the vector produced at a given position will in practice only depend on a small neighborhood.
- On the other hand, language often has long-range dependencies.
- For the model to leverage this knowledge, it has to remember the first material in the hidden state, update that hidden state repeatedly without forgetting what it learned.
- Instead we would like an architecture that can allow information to propagate more quickly and freely across the sequence. This architecture is the Transformer. But we need to introduce a few preliminaries first.

Attention mechanism



- This is a big slide.
- Attention is the core mechanism that allows us to model long-range dependencies effectively.
- Formally, attention takes a collection of input vectors x_1, \dots, x_n , a query vector y , and the goal is to process y in the context of the input vectors.
- Let's walk through the diagram slowly. We start with y and reduce its dimensionality. Similarly, we can reduce the dimensionality for all rows of \mathbf{x}^\top .
- Now the key part is to take the dot product between the representation of y and the representation of each x_i .
- We now have one column representing similarity scores (positive or negative) between y and each x_i .
- We apply the softmax, which exponentiates each component and then normalizes to 1.
- We can use the distribution to take a convex combination of the columns of \mathbf{x} . And finally we project onto the desired dimensionality.
- The above calculations is one attention head. In parallel, we go through the same motions to construct another vector. These are concatenated together and projected down to the original space.
- Another form of easy attention (called self-attention) is self-attention, in which case the queries are simply the same elements.
- This provides a sequence model where all pairs of elements of the sequence interact.

Layer normalization and residual connections

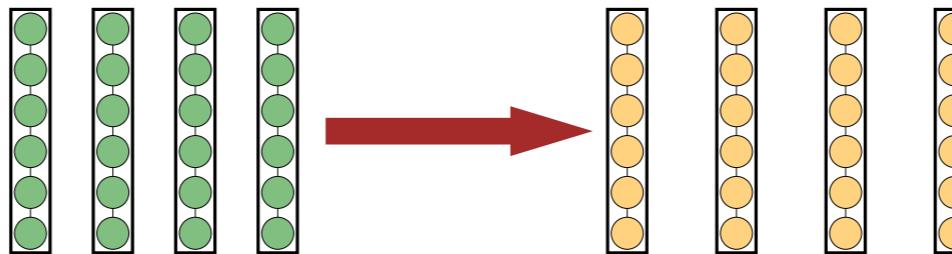


- There are two other pieces we need to introduce before we can fully define the Transformer: layer normalization and residual connections.
- They can be thought of as technical devices to make the final network easier to train.
- Together, they can be packed up into **AddNorm**. The interface is the same as a sequence model: it takes a sequence of vectors and produces a corresponding sequence of vectors.
- **AddNorm** takes a function f and applies it to the input \mathbf{x} . Then we add \mathbf{x} (called a residual connection), which allows information from \mathbf{x} to be directly passed through in case f is somehow messed up (say, early in training).
- Then we apply layer normalization, which takes a vector x and makes sure it's not too small or big (or else gradients might vanish or explode). Specifically, it subtracts the average of the elements of x and divides by the standard deviation. We do this for each vector in \mathbf{x} .
- In summary, **AddNorm** applies f to \mathbf{x} safely.

Transformer

-TransformerBlock : SequenceModel

$x_1 \ x_2 \ x_3 \ x_4$



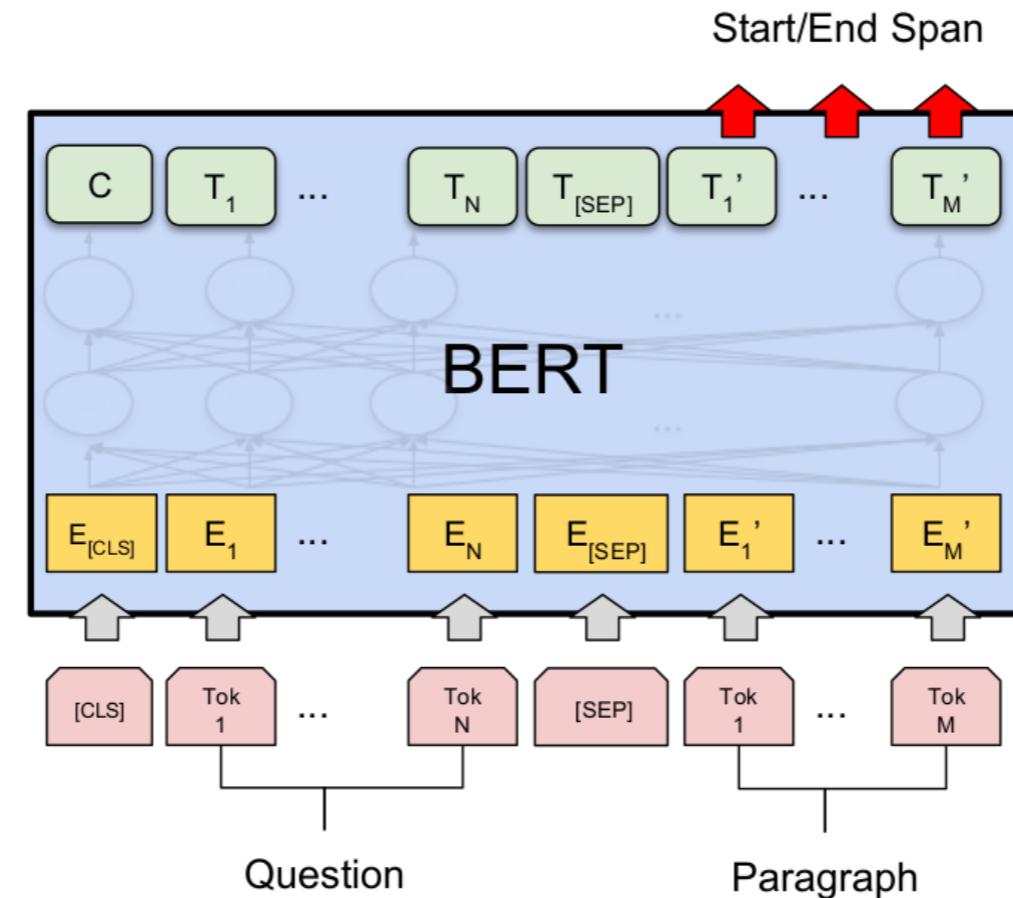
Processes each object x_i in context.

TransformerBlock(x) = AddNorm(FeedForward, AddNorm(Attention, x))

- Finally, we are ready to introduce the Transformer, which was introduced in 2017 and has really overthrown RNNs as the sequence model of choice.
- The **TransformerBlock** is a sequence model that takes a sequence of vectors to corresponding sequence of contextual vectors.
- We've already defined all the pieces, so the Transformer is just a one-liner.
- First, we apply self-attention to x to contextualize the vectors. Then we apply **AddNorm** (layer normalization with residual connections) to make things safe.
- Second, we apply a feedforward network to further process each vector independently. Then we do another **AddNorm**, and that's it.
- Note: in the actual Transformer, the **FeedForward** function has an extra matrix at the end.



BERT

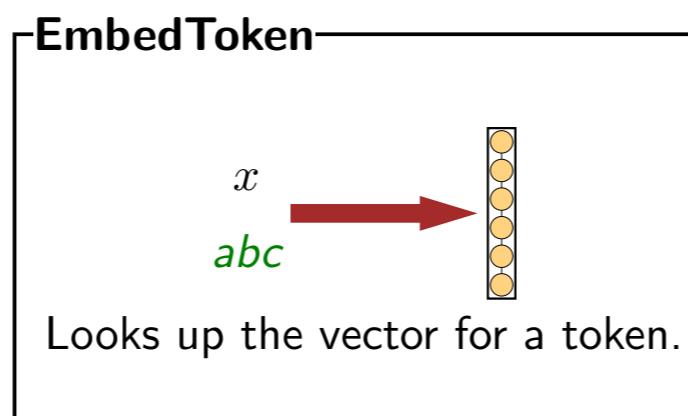
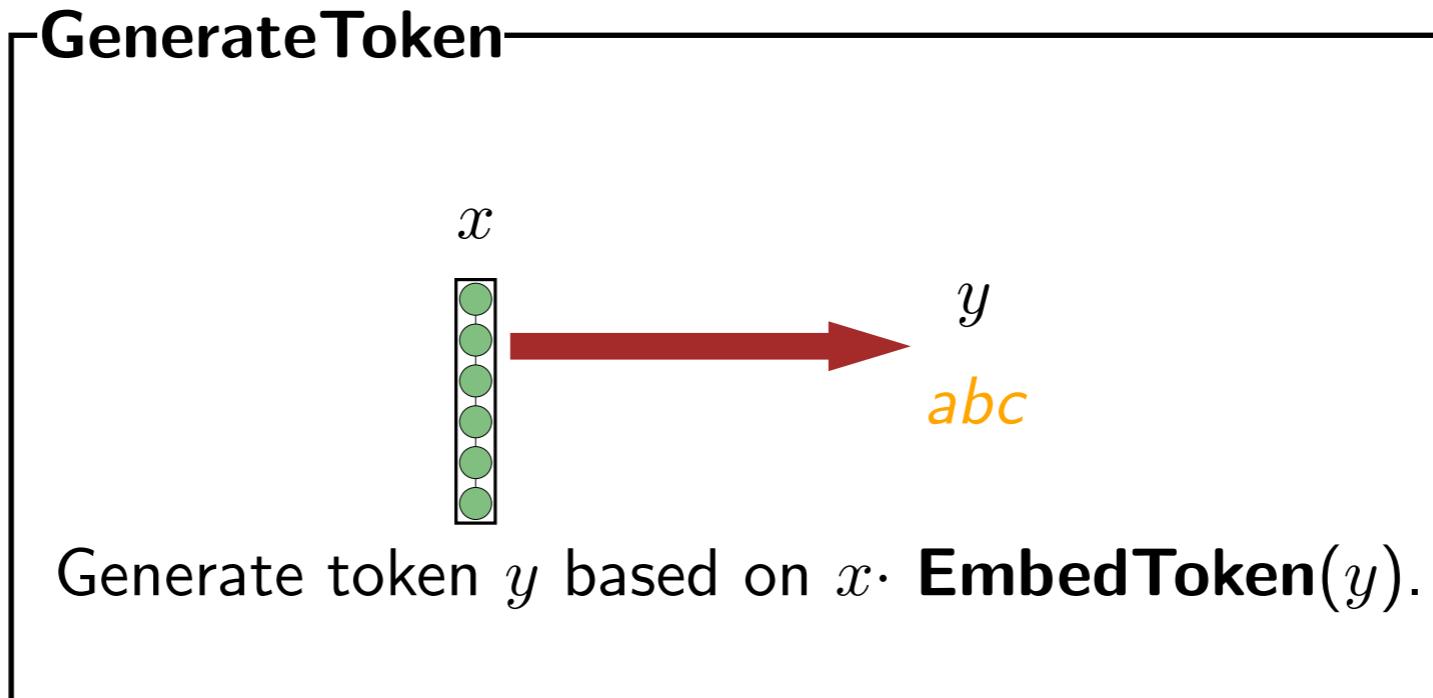


[CLS] What causes precipitation to fall? [SEP] In meterology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

$$\text{BERT}(\mathbf{x}) = \text{TransformerBlock}^{24}(\text{EmbedToken}(\mathbf{x}))$$

- Now we can explain BERT, the large unsupervised pretrained model which transformed NLP in 2018. Before then, there were many specialized architectures for different tasks but BERT was a single model architecture that worked well across many tasks.
- BERT is a function that takes a sentence, turns it into a sequence of vectors, and then just applies a Transformer block 24 times.
- When it's used for question answering, you concatenate the question with the paragraph.
- There are some details omitted: BERT defines tokens as word pieces as opposed to words. It also uses positional encodings as the Transformer block is invariant to the order of the vectors.
- Note that we are also not talking about the objective function (masked language modeling, next sentence prediction) used to train BERT.

Generating tokens



- So far, we've mostly studied how to design functions that can process a sentence (sequence of tokens or vectors).
- We can also generate new sequences.
- The basic building block for generation is something that takes a vector and outputs a token.
- This process is the reverse of **EmbedToken**, but uses it as follows: we compute a score $x \cdot \text{EmbedToken}(y)$ for each candidate token.
- Then we apply softmax (exponentiate and normalize) to get a distribution over words y . From this distribution, we can either take the token with the highest probability or simply sample a word from this distribution.

Generating sequences

LanguageModel

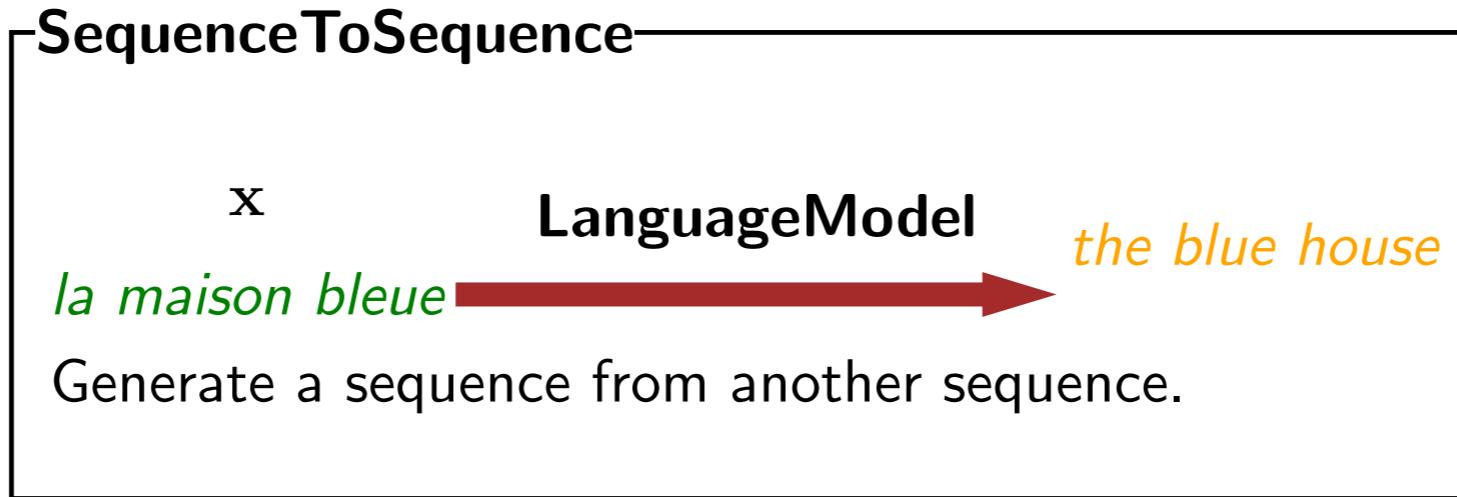


Generate next token in the sequence.

LanguageModel(x) = GenerateToken(Collapse(SequenceModel(EmbedToken(x))))

- In language modeling, we wish to generate (predict) the next token given the previous words generated so far.
- In this case, we simply take the history, which is a sequence of tokens, embed them, apply a sequence model (e.g., RNN or a Transformer).
- We then **Collapse** this into one vector (usually the last one).
- Now we've reduced the problem to something that **GenerateToken** can solve.

Sequence-to-sequence models



Applications:

- Machine translation: sentence to translation
- Document summarization: document to summary
- Semantic parsing: sentence to code

- Perhaps the most versatile interface is sequence-to-sequence models, where the model takes as input a sequence of tokens and produces an sequence of tokens.
- Importantly, the input sequence and the output sequence are not in 1:1 correspondence, and in general they have different lengths.
- Sequence-to-sequence models can be reduced to language modeling, where you feed in the input x along with the output tokens that you've generated so far, and ask for the next token.
- There are a number of applications (mostly in NLP) that uses sequence-to-sequence models: machine translation, document summarization, and semantic parsing to name a few.



Summary

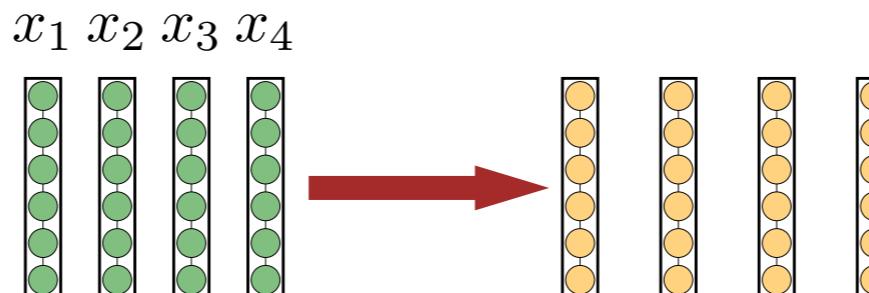
FeedForward Conv MaxPool

EmbedToken SequenceRNN SimpleRNN LSTM

Attention AddNorm TransformerBlock BERT

Collapse GenerateToken LanguageModel SequenceToSequence

SequenceModel



Process each element of a sequence with respect to other elements.

- In summary, we've done a whirlwind tour over different types of differentiable programs from deep learning.
- We started with feedforward networks (**FeedForward**), and then talked about convolutional neural networks (**Conv**, **MaxPool**).
- Then we considered token sequences (e.g., text), where we always start by embedding the tokens into vectors (**EmbedToken**).
- There are two paths: The first uses RNNs and the second uses Transformers.
- There are many details that we have glossed over, but the thing you should take away is to understand rigorously what the type signature of all these functions are, and some intuition behind what the function is meant to be doing.



Machine learning: generalization



- In this module, I will talk about the generalization of machine learning algorithms.

Minimizing training loss

Hypothesis class:

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

Training objective (loss function):

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

Optimization algorithm:

stochastic gradient descent

Is the training loss a good objective to optimize?

- Recall that our machine learning framework consists of specifying the hypothesis class, loss function, and the optimization algorithm.
- The hypothesis class could be linear predictors or neural networks. The loss function could be the hinge loss or the squared loss, which is averaged to produce the training loss.
- The default optimization algorithm is (stochastic) gradient descent.
- But let's be a bit more critical about the training loss. Is the training loss the right thing to optimize?



A strawman algorithm



Algorithm: rote learning

Training: just store $\mathcal{D}_{\text{train}}$.

Predictor $f(x)$:

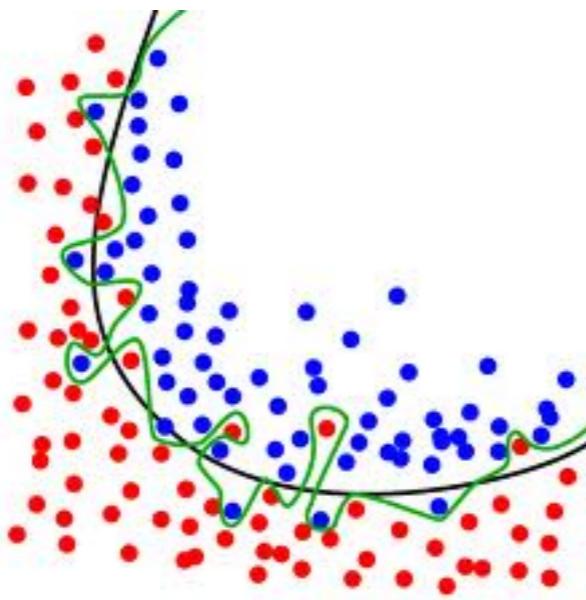
If $(x, y) \in \mathcal{D}_{\text{train}}$: return y .

Else: **segfault**.

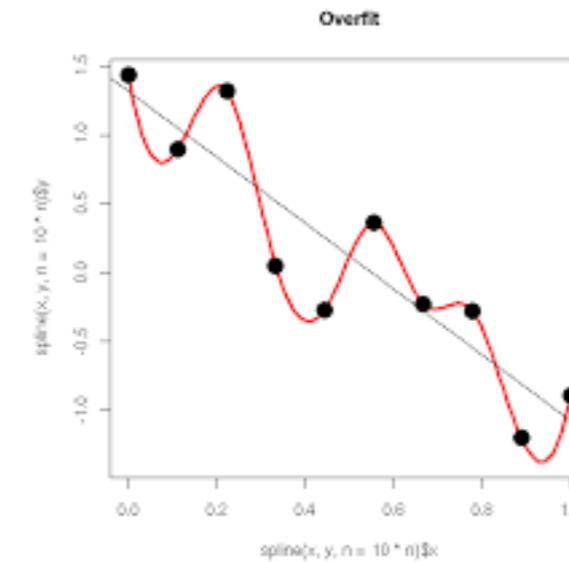
Minimizes the objective perfectly (zero), but clearly bad...

- Here is a strategy to consider: the rote learning algorithm, which just memorizes the training data and crashes otherwise.
- The rote learning algorithm does a perfect job of minimizing the training loss.
- But it's clearly a bad idea: It **overfits** to the training data and doesn't **generalize** to unseen examples.
- So clearly machine learning can't be about just minimizing the training loss.

Overfitting pictures



Classification



Regression

- This is an extreme example of **overfitting**, which is when a learning algorithm outputs a predictor that does well on the training data but not well on new examples.
- Here are two pictures that illustrate what overfitting looks like for binary classification and regression.
- On the left, we see that the green decision boundary gets zero training loss by separating all the blue points from the red ones. However, the smoother and simpler black curve is intuitively more likely to be the better classifier.
- On the right, we see that the predictor that goes through all the points will get zero training loss, but intuitively, the black line is perhaps a better option.
- In both cases, what is happening is that by over-optimizing on the training set, we risk fitting **noise** in the data.

Evaluation



How good is the predictor f ?



Key idea: the real learning objective

Our goal is to minimize **error on unseen future examples**.

Don't have unseen examples; next best thing:



Definition: test set

Test set $\mathcal{D}_{\text{test}}$ contains examples not used for training.

- So what is the true objective then? Taking a step back, machine learning is just a means to an end. What we're really doing is building a predictor to be deployed in the real world, and we just happen to be using machine learning. What we really care about is how accurate that predictor is on those **unseen future** inputs.
- Of course, we can't access unseen future examples, so the next best thing is to create a **test set**. As much as possible, we should treat the test set as a pristine thing that's unseen. We definitely should not tune our predictor based on the test set, because we wouldn't be able to do that on future examples.
- Of course at some point we have to run our algorithm on the test set, but just be aware that each time this is done, the test set becomes less good of an indicator of how well your predictor is actually doing.

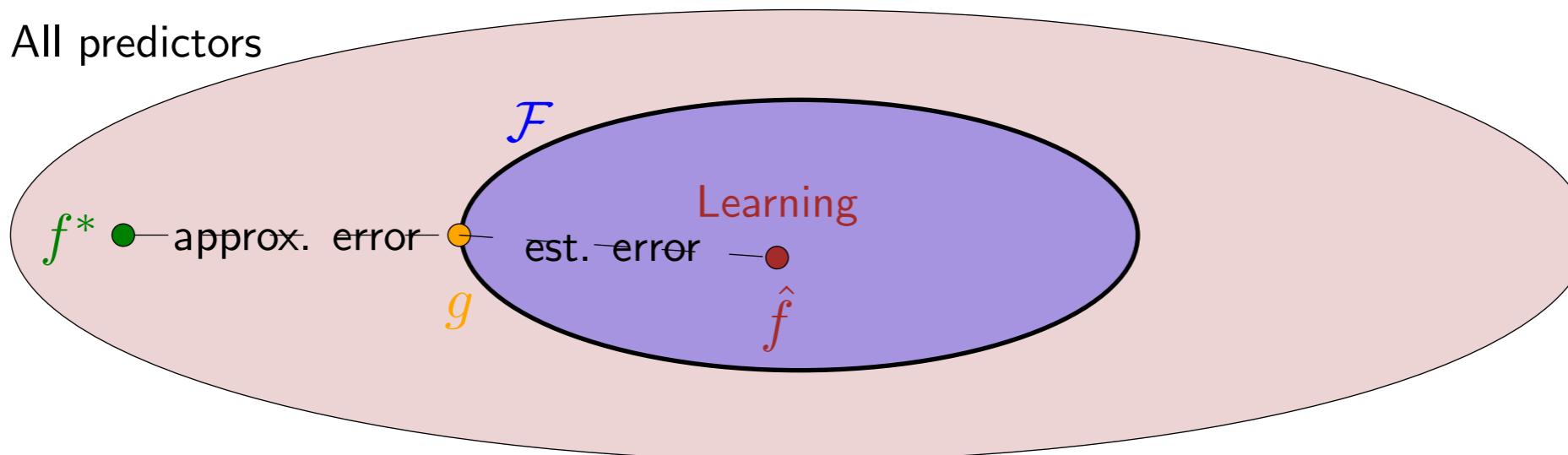
Generalization

When will a learning algorithm **generalize** well?



- So far, we have an intuitive feel for what overfitting is. How do we make this precise? In particular, when does a learning algorithm generalize from the training set to the test set?

Approximation and estimation error

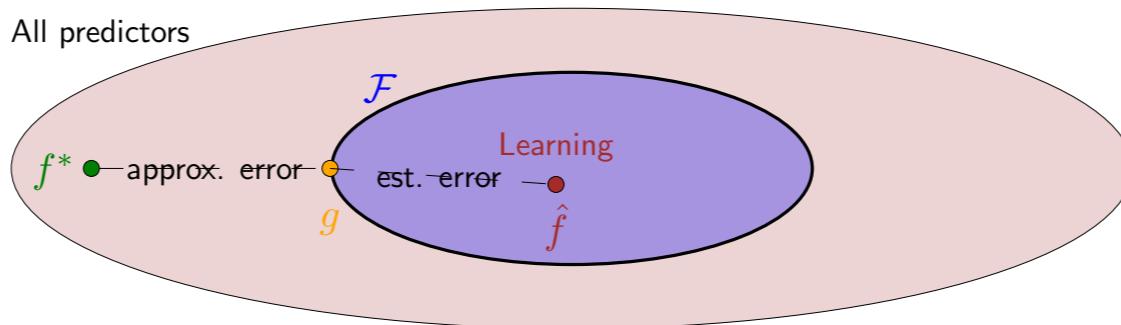


- Approximation error: how good is the hypothesis class?
- Estimation error: how good is the learned predictor **relative to** the potential of the hypothesis class?

$$\text{Err}(\hat{f}) - \text{Err}(f^*) = \underbrace{\text{Err}(\hat{f}) - \text{Err}(g)}_{\text{estimation}} + \underbrace{\text{Err}(g) - \text{Err}(f^*)}_{\text{approximation}}$$

- Here's a cartoon that can help you understand the balance between fitting and generalization. Out there somewhere, there is a magical predictor f^* that classifies everything perfectly. This predictor is unattainable; all we can hope to do is to use a combination of our domain knowledge and data to approximate that. The question is: how far are we away from f^* ?
- Recall that our learning framework consists of (i) choosing a hypothesis class \mathcal{F} (e.g., by defining the feature extractor) and then (ii) choosing a particular predictor \hat{f} from \mathcal{F} .
- **Approximation error** is how far the entire hypothesis class is from the target predictor f^* . Larger hypothesis classes have lower approximation error. Let $g \in \mathcal{F}$ be the best predictor in the hypothesis class in the sense of minimizing test error $g = \arg \min_{f \in \mathcal{F}} \text{Err}(f)$. Here, distance is just the differences in test error: $\text{Err}(g) - \text{Err}(f^*)$.
- **Estimation error** is how good the predictor \hat{f} returned by the learning algorithm is with respect to the best in the hypothesis class: $\text{Err}(\hat{f}) - \text{Err}(g)$. Larger hypothesis classes have higher estimation error because it's harder to find a good predictor based on limited data.
- We'd like both approximation and estimation errors to be small, but there's a tradeoff here.

Effect of hypothesis class size



As the hypothesis class size increases...

Approximation error decreases because:

taking min over larger set

Estimation error increases because:

harder to estimate something more complex

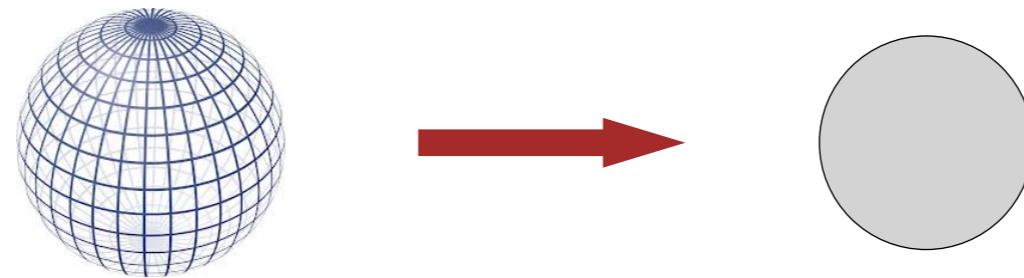
How do we control the hypothesis class size?

- The approximation error decreases monotonically as the hypothesis class size increases for a simple reason: you're taking a minimum over a larger set.
- The estimation error increases monotonically as the hypothesis class size increases for a deeper reason involving statistical learning theory (explained in CS229T).

Strategy 1: dimensionality

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

Reduce the dimensionality d (number of features):



- Let's focus our attention to linear predictors. For each weight vector \mathbf{w} , we have a predictor $f_{\mathbf{w}}$ (for classification, $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$). So the hypothesis class $\mathcal{F} = \{f_{\mathbf{w}}\}$ is all the predictors as \mathbf{w} ranges. By controlling the number of possible values of \mathbf{w} that the learning algorithm is allowed to choose from, we control the size of the hypothesis class and thus guard against overfitting.
- One straightforward strategy is to change the dimensionality, which is the number of features. For example, linear functions are lower-dimensional than quadratic functions.

Controlling the dimensionality

Manual feature (template) selection:

- Add feature templates if they help
- Remove feature templates if they don't help

Automatic feature selection (beyond the scope of this class):

- Forward selection
- Boosting
- L_1 regularization

It's the number of features that matters

- Mathematically, you can think about removing a feature $\phi(x)_{37}$ as simply only allowing its corresponding weight to be zero ($w_{37} = 0$).
- Operationally, if you have a few feature templates, then it's probably easier to just manually include or exclude them — this will give you more intuition.
- If you have a lot of individual features, you can apply more automatic methods for selecting features, but these are beyond the scope of this class.
- An important point is that it's the number of features that matters, not the number of feature templates. (Can you define one feature template that results in severe overfitting?) Nor is it the amount of code that you have to write to generate a particular feature.

Strategy 2: norm

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

Reduce the norm (length) $\|\mathbf{w}\|$:



- A related way to keep the weights small is called **regularization**, which involves adding an additional term to the objective function which penalizes the norm (length) of w . This is probably the most common way to control the norm.

Controlling the norm

Regularized objective:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|^2$$



Algorithm: gradient descent

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

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

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

Same as gradient descent, except shrink the weights towards zero by λ .

- This form of regularization is also known as L_2 regularization, or weight decay in deep learning literature.
- We can use gradient descent on this regularized objective, and this simply leads to an algorithm which subtracts a scaled down version of \mathbf{w} in each iteration. This has the effect of keeping \mathbf{w} closer to the origin than it otherwise would be.
- Note: Support Vector Machines are exactly hinge loss + L_2 regularization.

Controlling the norm: early stopping



Algorithm: gradient descent

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

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

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

Idea: simply make T smaller

Intuition: if have fewer updates, then $\|\mathbf{w}\|$ can't get too big.

Lesson: try to minimize the training error, but don't try too hard.

- A really cheap way to keep the weights small is to do **early stopping**. As we run more iterations of gradient descent, the objective function improves. If we cared about the objective function, this would always be a good thing. However, our true objective is not the training loss.
- Each time we update the weights, w has the potential of getting larger, so by running gradient descent a fewer number of iterations, we are implicitly ensuring that w stays small.
- Though early stopping seems hacky, there is actually some theory behind it. And one paradoxical note is that we can sometimes get better solutions by performing less computation.



Summary

Not the real objective: training loss

Real objective: loss on unseen future examples

Semi-real objective: test loss



Key idea: keep it simple

Try to minimize training error, but keep the hypothesis class small.



- In summary, we started by noting that the training loss is not the objective. Instead it is minimizing unseen future examples, which is approximated by the test set provided you are careful.
- We've seen several ways to control the size of the hypothesis class (and thus reducing the estimation error) based on either reducing the dimensionality or reducing the norm.
- It is important to note that what matters is the **size** of the hypothesis class, not how "complex" the predictors in the hypothesis class look. To put it another way, using complex features backed by 1000 lines of code doesn't hurt you if there are only 5 of them.
- So far, we've talked about the various knobs that we can turn to control the size of the hypothesis class, but how much do we turn each knob?



Machine learning: best practices



- We've spent a lot of talking about the formal principles of machine learning.
- In this module, I will discuss some of the more empirical aspects you encounter in practice.



Choose your own adventure

Hypothesis class:

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

Feature extractor ϕ : linear, quadratic

Architecture: number of layers, number of hidden units

Training objective:

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

Loss function: hinge, logistic

Regularization: none, L2

Optimization algorithm:



Algorithm: stochastic gradient descent

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

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

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

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

Number of epochs

Step size: constant, decreasing, adaptive

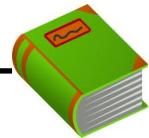
Initialization: amount of noise, pre-training

Batch size

Dropout

- Recall that there are three design decisions for setting up a machine learning algorithm: the hypothesis class, the training objective, and the optimization algorithm.
- For the hypothesis class, there are two knobs you can turn. The first is the feature extractor ϕ (linear features, quadratic features, indicator features on regions, etc. The second is the architecture of the predictor: linear (one layer) or neural network with layers, and in the case of neural networks, how many hidden units (k) do we have.
- The second design decision is to specify the training objective, which we do by specifying the loss function depending how we want the predictor to fit our data, and also whether we want to regularize the weights to guard against overfitting.
- The final design decision is how to optimize the predictor. Even the basic stochastic gradient descent algorithm has at least two knobs: how long to train (number of epochs) and how aggressively to update (the step size). On top of that are many enhancements and tweaks common to training deep neural networks: changing the step size over time, perhaps adaptively, how we initialize the weights, whether we update on batches (say of 16 examples) instead of 1, and whether we apply dropout to guard against overfitting.
- So it is really a choose your own machine learning adventure. Sometimes decisions can be made via prior knowledge and are thoughtful (e.g., features that capture periodic trends). But in many (even most) cases, we don't really know what the proper values should be. Instead, we want a way to have these just set automatically.

Hyperparameters



Definition: hyperparameters

Design decisions (hypothesis class, training objective, optimization algorithm) that need to be made before running the learning algorithm.

How do we choose hyperparameters?

Choose hyperparameters to minimize $\mathcal{D}_{\text{train}}$ error?

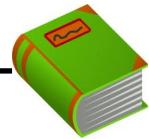
No - optimum would be to include all features, no regularization, train forever

Choose hyperparameters to minimize $\mathcal{D}_{\text{test}}$ error?

No - choosing based on $\mathcal{D}_{\text{test}}$ makes it an unreliable estimate of error!

- Each of these many design decisions is a **hyperparameter**.
- We could choose the hyperparameters to minimize the training loss. However, this would lead to a degenerate solution. For example, by adding additional features, we can always decrease the training loss, so we would just end up adding all the features in the world, leading to a model that wouldn't generalize. We would turn off all regularization, because that just gets in the way of minimizing the training loss.
- What if we instead chose hyperparameters to minimize the test loss. This might lead to good hyperparameters, but is problematic because you then lose the ability to measure how well you're doing. Recall that the test set is supposed to be a surrogate for unseen examples, and the more you optimize over them, the less unseen they become.

Validation set



Definition: validation set

A **validation set** is taken out of the training set and used to optimize hyperparameters.



For each setting of hyperparameters, train on $\mathcal{D}_{\text{train}} \setminus \mathcal{D}_{\text{val}}$, evaluate on \mathcal{D}_{val}

- The solution is to invent something that looks like a test set. There's no other data lying around, so we'll have to steal it from the training set. The resulting set is called the **validation set**.
- The size of the validation set should be large enough to give you a reliable estimate, but you don't want to take away too many examples from the training set.
- With this validation set, now we can simply try out a bunch of different hyperparameters and choose the setting that yields the lowest error on the validation set. Which hyperparameter values should we try? Generally, you should start by getting the right order of magnitude (e.g., $\lambda = 0.0001, 0.001, 0.01, 0.1, 1, 10$) and then refining if necessary.
- In K -fold **cross-validation**, you divide the training set into K parts. Repeat K times: train on $K - 1$ of the parts and use the other part as a validation set. You then get K validation errors, from which you can compute and report both the mean and the variance, which gives you more reliable information.

Model development strategy

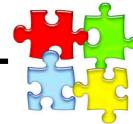


Algorithm: Model development strategy

- Split data into train, validation, test
- Look at data to get intuition
- Repeat:
 - Implement model/feature, adjust hyperparameters
 - Run learning algorithm
 - Sanity check train and validation error rates
 - Look at weights and prediction errors
- Evaluate on test set to get final error rates

- This slide represents the most important yet most overlooked part of machine learning: how to actually apply it in practice.
- We have so far talked about the mathematical foundation of machine learning (loss functions and optimization), and discussed some of the conceptual issues surrounding overfitting, generalization, and the size of hypothesis classes. But what actually takes most of your time is not writing new algorithms, but going through a **development cycle**, where you iteratively improve your system.
- The key is to stay connected with the data and the model, and have intuition about what's going on. Make sure to empirically examine the data before proceeding to the actual machine learning. It is imperative to understand the nature of your data in order to understand the nature of your problem.
- First, maintain data hygiene. Hold out a test set from your data that you don't look at until you're done. Start by looking at the (training or validation) data to get intuition. You can start to brainstorm what features / predictors you will need. You can compute some basic statistics.
- Then you enter a loop: implement a new model architecture or feature template. There are three things to look at: error rates, weights, and predictions. First, sanity check the error rates and weights to make sure you don't have an obvious bug. Then do an **error analysis** to see which examples your predictor is actually getting wrong. The art of practical machine learning is turning these observations into new features.
- Finally, run your system once on the test set and report the number you get. If your test error is much higher than your validation error, then you probably did too much tweaking and were **overfitting** (at a meta-level) the validation set.

Model development strategy example



Problem: simplified named-entity recognition

Input: a string x (e.g., *Governor Gavin Newsom in*)

Output: y , whether x (excluding first/last word) is a person or not (e.g., +1)

[code]

- Let's try out the model development strategy on the task of training a classifier to predict whether a string is person or not (excluding the first and last context words).
- First, let us look at the data (`names.train`). Starting simple, we define the empty feature template, which gets horrible error.
- Then we define a single feature template "entity is ____". Look at `weights` (person names have positive weight, city names have negative weight) and `error-analysis`.
- Let us add "left is ____" and "right is ____" feature templates based on the errors (e.g., the word "said" is indicative of a person). Look at `weights` ("the" showing up on the left indicates not a person) and `error-analysis`.
- Let us add feature templates "entity contains ____". Look at `weights` and `error-analysis`.
- Let us add feature templates "entity contains prefix ____" and "entity contains suffix ____". Look at `weights` and `error-analysis`.
- Finally we run it on the test set.

Tips



Start simple:

- Run on small subsets of your data or synthetic data
- Start with a simple baseline model
- Sanity check: can you overfit 5 examples

Log everything:

- Track training loss and validation loss over time
- Record hyperparameters, statistics of data, model, and predictions
- Organize experiments (each run goes in a separate folder)

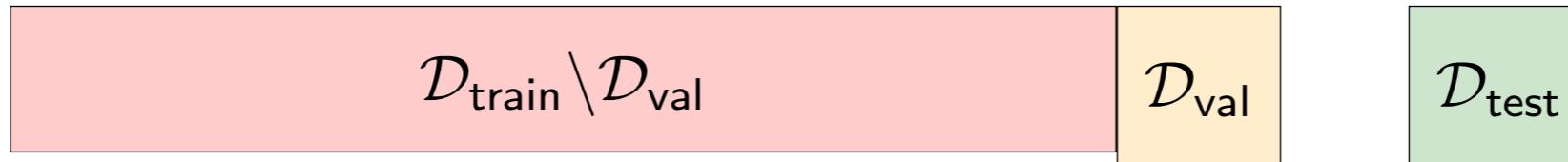
Report your results:

- Run each experiment multiple times with different random seeds
- Compute multiple metrics (e.g., error rates for minority groups)

- There is more to be said about the practice of machine learning. Here are some pieces of advice. Note that many related to simply good software engineering practices.
- First, don't start out by coding up a large complex model and try running it on a million examples. Start simple, both with the data (small number of examples) and the model (e.g., linear classifier). Sanity check that things are working first before increasing the complexity. This will help you debug in a regime where things are more interpretable and also things run faster. One sanity check is to train a sufficiently expressive model on a few very examples and see if the model can overfit the examples (get zero training error). This does not produce a useful model, but is a diagnostic to see if the optimization is working. If you can't overfit on 5 examples, then you have a problem: maybe the hypothesis class is too small, the data is too noisy, or the optimization isn't working.
- Second, log everything so you can diagnose problems. Monitor the losses over epochs. It is also important to track the training loss so that if you get bad results, is it due to bad optimization or overfitting. Record all the hyperparameters, so that you have a full record of how to reproduce the results.
- Third, when you report your results, you should be able to run an experiment multiple times with different randomness to see how stable the results are. Report error bars. And finally, if it makes sense for your application to report more than just a single test accuracy. Report the errors for minority groups and add if your model is treating every group fairly.



Summary



Don't look at the test set!

Understand the data!

Start simple!

Practice!

- To summarize, we've talked about the practice of machine learning.
- First, make sure you follow good data hygiene, separating out the test set and don't look at it.
- But you should look at the training or validation set to get intuition about your data before you start.
- Then, start simple and make sure you understand how things are working.
- Beyond that, there are a lot of design decisions to be made (hyperparameters). So the most important thing is to practice, so that you can start developing more intuition, and developing a set of best practices that works for you.



Machine learning: k-means



- In this module, we'll talk about **K-means**, a simple algorithm for clustering, a form of unsupervised learning.

Word clustering

Input: raw text (100 million words of news articles)...

Output:

Cluster 1: Friday Monday Thursday Wednesday Tuesday Saturday Sunday weekends Sundays Saturdays

Cluster 2: June March July April January December October November September August

Cluster 3: water gas coal liquid acid sand carbon steam shale iron

Cluster 4: great big vast sudden mere sheer gigantic lifelong scant colossal

Cluster 5: man woman boy girl lawyer doctor guy farmer teacher citizen

Cluster 6: American Indian European Japanese German African Catholic Israeli Italian Arab

Cluster 7: pressure temperature permeability density porosity stress velocity viscosity gravity tension

Cluster 8: mother wife father son husband brother daughter sister boss uncle

Cluster 9: machine device controller processor CPU printer spindle subsystem compiler plotter

Cluster 10: John George James Bob Robert Paul William Jim David Mike

Cluster 11: anyone someone anybody somebody

Cluster 12: feet miles pounds degrees inches barrels tons acres meters bytes

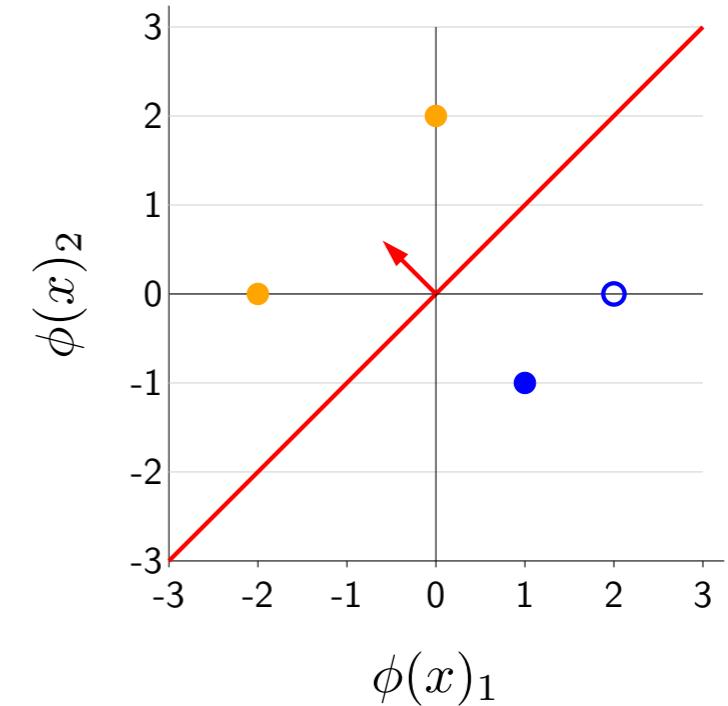
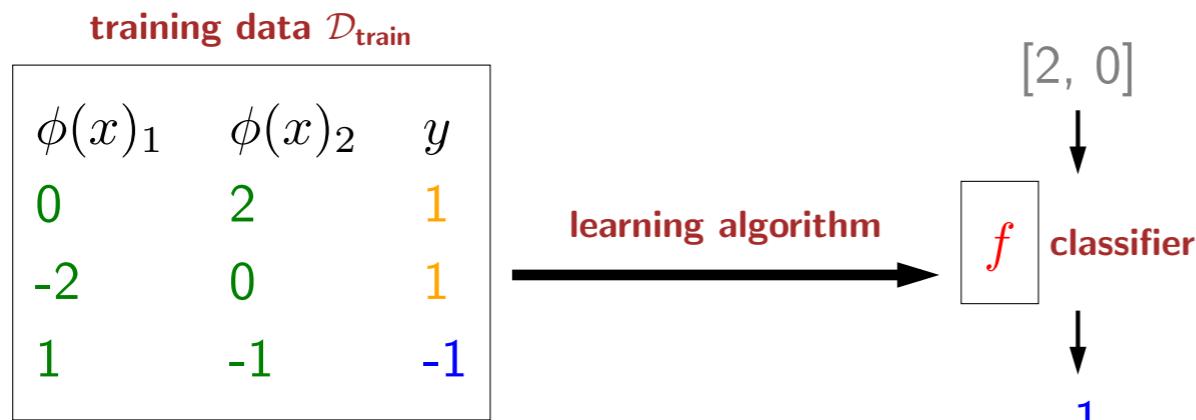
Cluster 13: director chief professor commissioner commander treasurer founder superintendent dean custodian

Cluster 14: had hadn't hath would've could've should've must've might've

Cluster 15: head body hands eyes voice arm seat eye hair mouth

- Here is a classic example of clustering from the NLP literature, called Brown clustering. This was the unsupervised learning method of choice before word vectors.
- The input to the algorithm is simply raw text, and the output is a clustering of the words.
- The first cluster more or less represents days of the week, the second is months, the third is natural resources, and so on.
- It is important to note that no one told the algorithm what days of the week were or months or family relations. The clustering algorithm discovered this structure automatically.
- On a personal note, Brown clustering was actually my first experience that got me to pursue research in NLP. Seeing the results of unsupervised learning when it works was just magical. And of course today, we're seeing even more strongly the potential of unsupervised learning with neural language models such as BERT and GPT-3.

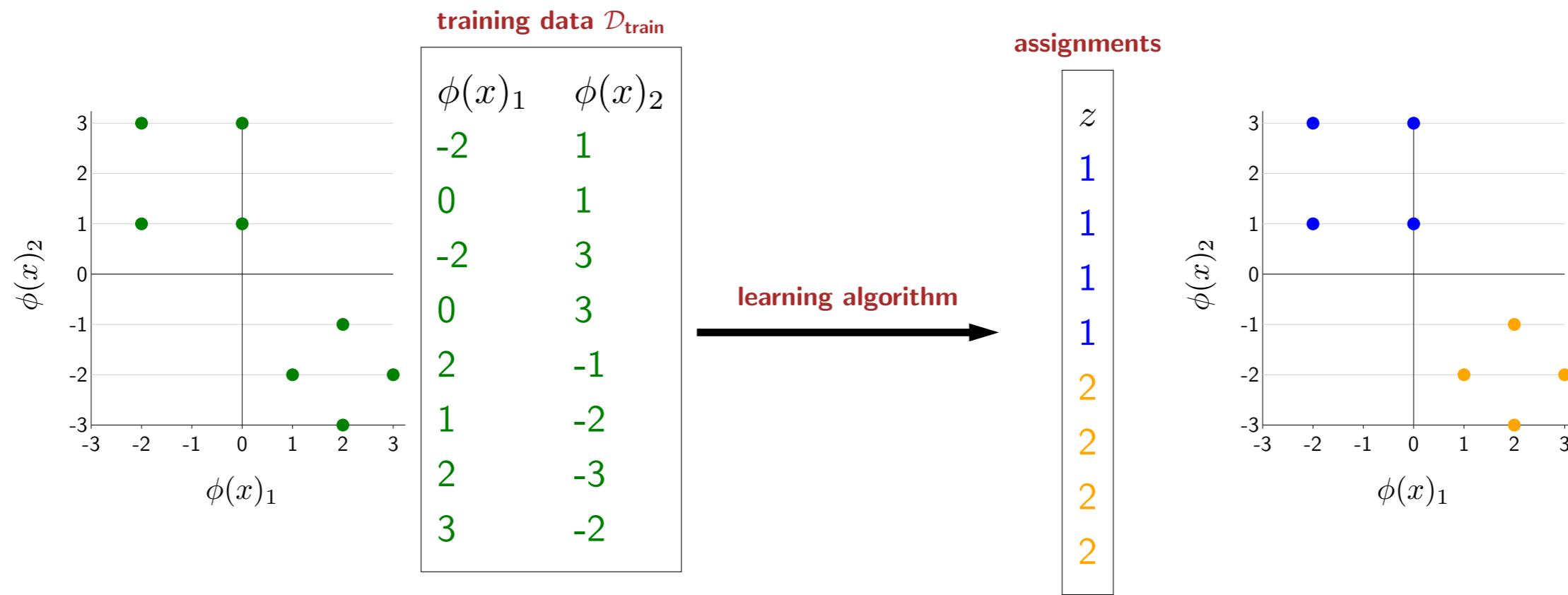
Classification (supervised learning)



Labeled data is expensive to obtain

- I want to contrast unsupervised learning with supervised learning.
- Recall that in classification you're given a set of **labeled** training examples.
- A learning algorithm produces a classifier that can classify new points.
- Note that we're now plotting the (two-dimensional) feature vector rather than the raw input, since the learning algorithms only depend on the feature vectors.
- However, the main challenge with supervised learning is that it can be expensive to collect the labels for data.

Clustering (unsupervised learning)



Intuition: Want to assign nearby points to same cluster

Unlabeled data is very cheap to obtain

- In contrast, in clustering, you are only given **unlabeled** training examples.
- Our goal is to assign each point to a cluster. In this case, there are two clusters, 1 (blue) and 2 (orange).
- Intuitively, nearby points should be assigned to the same cluster.
- The advantage of unsupervised learning is that unlabeled data is often very cheap and almost free to obtain, especially text or images on the web.

Clustering task



Definition: clustering

Input: training points

$$\mathcal{D}_{\text{train}} = [x_1, \dots, x_n]$$

Output: assignment of each point to a cluster

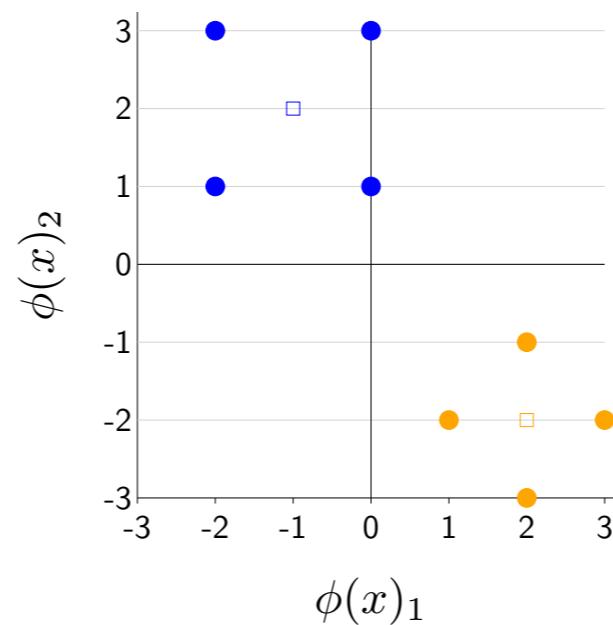
$$\mathbf{z} = [z_1, \dots, z_n] \text{ where } z_i \in \{1, \dots, K\}$$

- Formally, the task of clustering is to take a set of points as input and return a partitioning of the points into K clusters.
- We will represent the partitioning using an **assignment vector** $\mathbf{z} = [z_1, \dots, z_n]$.
- For each i , $z_i \in \{1, \dots, K\}$ specifies which of the K clusters point i is assigned to.

Centroids

Each cluster $k = 1, \dots, K$ is represented by a **centroid** $\mu_k \in \mathbb{R}^d$

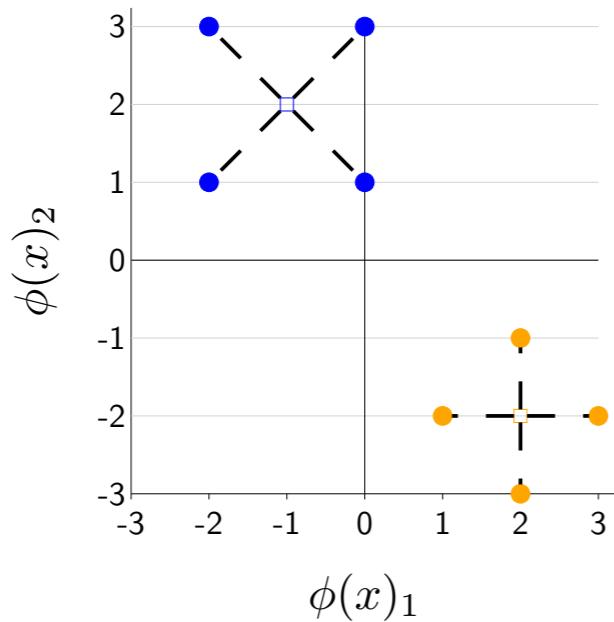
$$\boldsymbol{\mu} = [\mu_1, \dots, \mu_K]$$



Intuition: want each point $\phi(x_i)$ to be close to its assigned centroid μ_{z_i}

- What makes a cluster? The key assumption is that each cluster k is represented by a **centroid** μ_k .
- Now the intuition is that we want each point $\phi(x_i)$ to be close to its assigned centroid μ_{z_i} .

K-means objective



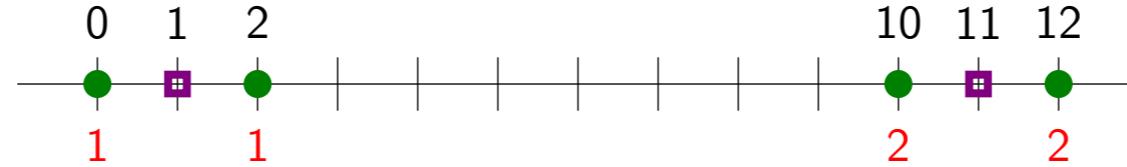
$$\text{Loss}_{\text{kmeans}}(\mathbf{z}, \boldsymbol{\mu}) = \sum_{i=1}^n \|\phi(x_i) - \boldsymbol{\mu}_{\mathbf{z}_i}\|^2$$

$$\min_{\mathbf{z}} \min_{\boldsymbol{\mu}} \text{Loss}_{\text{kmeans}}(\mathbf{z}, \boldsymbol{\mu})$$

- To formalize this, we define the K-means objective (distinct from the K-means algorithm).
- The variables are the assignments \mathbf{z} and centroids $\boldsymbol{\mu}$.
- We examine the squared distance (dashed lines) from a point $\phi(x_i)$ to the centroid of its assigned cluster μ_{z_i} . Summing over all these squared distances gives the K-means objective.
- This loss can be interpreted as a reconstruction loss: imagine replacing each data point by its assigned centroid. Then the objective captures how lossy this compression was.
- Now our goal is to minimize the K-means loss.



Alternating minimization from optimum



If know centroids $\mu_1 = 1$, $\mu_2 = 11$:

$$z_1 = \arg \min \{(0 - 1)^2, (0 - 11)^2\} = 1$$

$$z_2 = \arg \min \{(2 - 1)^2, (2 - 11)^2\} = 1$$

$$z_3 = \arg \min \{(10 - 1)^2, (10 - 11)^2\} = 2$$

$$z_4 = \arg \min \{(12 - 1)^2, (12 - 11)^2\} = 2$$

If know assignments $z_1 = z_2 = 1$, $z_3 = z_4 = 2$:

$$\mu_1 = \arg \min_{\mu} (0 - \mu)^2 + (2 - \mu)^2 = 1$$

$$\mu_2 = \arg \min_{\mu} (10 - \mu)^2 + (12 - \mu)^2 = 11$$

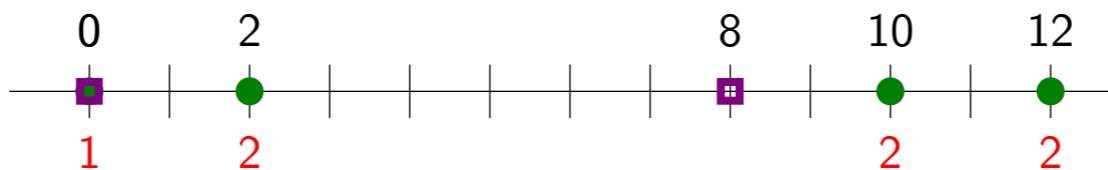
- Before we present the K-means algorithm, let us form some intuitions.
- Consider the following one-dimensional clustering problem with 4 points. Intuitively there are two clusters.
- Suppose we know the centroids. Then for each point the assignment that minimizes the K-means loss is the closer of the two centroids.
- Suppose we know the assignments. Then for each cluster, we average the points that are assigned to that cluster.

Alternating minimization from random initialization

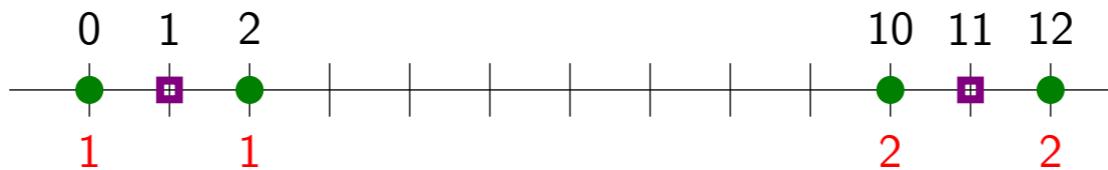
Initialize μ :



Iteration 1:



Iteration 2:



Converged.

- But of course we don't know either the centroids or assignments.
- So we simply start with an arbitrary setting of the centroids.
- Then alternate between choosing the best assignments given the centroids, and choosing the best centroids given the assignments.
- This is the K-means algorithm.

K-means algorithm



Algorithm: K-means

Initialize $\mu = [\mu_1, \dots, \mu_K]$ randomly.

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

 Step 1: set assignments \mathbf{z} given μ

 For each point $i = 1, \dots, n$:

$$z_i \leftarrow \arg \min_{k=1, \dots, K} \|\phi(x_i) - \mu_k\|^2$$

 Step 2: set centroids μ given \mathbf{z}

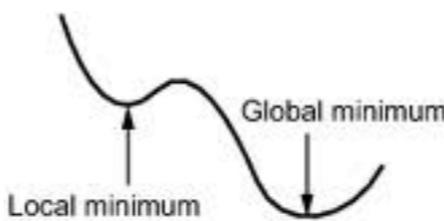
 For each cluster $k = 1, \dots, K$:

$$\mu_k \leftarrow \frac{1}{|\{i : z_i = k\}|} \sum_{i:z_i=k} \phi(x_i)$$

- Now we can state the K-means algorithm formally. We start by initializing all the centroids randomly. Then, we iteratively alternate back and forth between steps 1 and 2, optimizing \mathbf{z} given $\boldsymbol{\mu}$ and vice-versa.
- **Step 1** of K-means fixes the centroids $\boldsymbol{\mu}$. Then we can optimize the K-means objective with respect to \mathbf{z} alone quite easily. It is easy to show that the best label for z_i is the cluster k that minimizes the distance to the centroid μ_k (which is fixed).
- **Step 2** turns things around and fixes the assignments \mathbf{z} . We can again look at the K-means objective function and optimize it with respect to the centroids $\boldsymbol{\mu}$. The best μ_k is to place the centroid at the average of all the points assigned to cluster k .

Local minima

K-means is guaranteed to converge to a local minimum, but is not guaranteed to find the global minimum.



[demo: getting stuck in local optima, seed = 100]

Solutions:

- Run multiple times from different random initializations
- Initialize with a heuristic (K-means++)

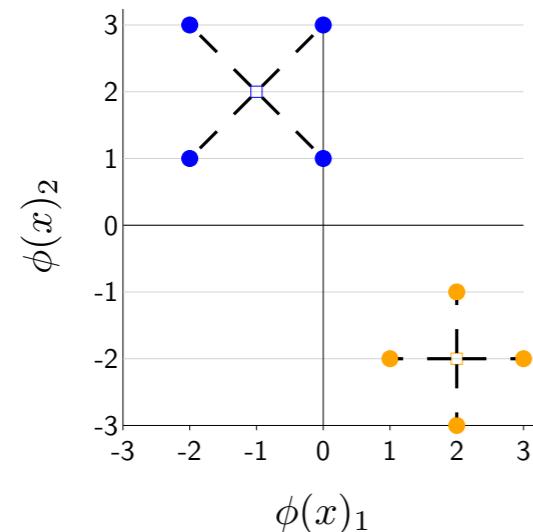
- K-means is guaranteed to decrease the loss function each iteration and will converge to a local minimum, but it is not guaranteed to find the global minimum, so one must exercise caution when applying K-means.
- Advanced: One solution is to simply run K-means several times from multiple random initializations and then choose the solution that has the lowest loss.
- Advanced: Or we could try to be smarter in how we initialize K-means. K-means++ is an initialization scheme which places centroids on training points so that these centroids tend to be distant from one another.



Summary

Clustering: discover structure in unlabeled data

K-means objective:



K-means algorithm:



assignments \mathbf{z}

centroids $\boldsymbol{\mu}$

Unsupervised learning use cases:

- Data exploration and discovery
- Providing representations to downstream supervised learning

- In summary, K-means is a simple and widely-used method for discovering cluster structure in data.
- Note that K-means can mean two things: the objective and the algorithm.
- Given points we define the K-means objective as the sum of the squared differences between a point and its assigned centroid.
- We also defined the K-means algorithm, which performs alternating optimization on the K-means objective.
- Finally, clustering is just one instance of unsupervised learning, which seeks to learn models from the wealth of unlabeled data alone. Unsupervised learning can be used in two ways: exploring a dataset which has not been labeled (let the data speak), and learning representations (discrete clusters or continuous embeddings) useful for downstream supervised applications.

Examples: [class] [email] [regression] [noise] [nonlinear] [cluster] [new]
[Background] [Documentation]

Step (or press ctrl-enter in text box)

Algorithm: **stochastic gradient descent on Loss_{hinge}**(x, y, \mathbf{w}) = $\max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$

Train error: **4/4 = 1**

Start of algorithm. Click "Step" to step through it.

Learning algorithms

- An example is an input-output pair (x, y) . The input could be anything (strings, images, videos). In principle the output could be anything, but we will focus on the case where the output $y \in \{+1, -1\}$ (binary classification) and $y \in \mathbb{R}$ (regression).
- We are given a set of training examples $\mathcal{D}_{\text{train}}$ and a set of test examples $\mathcal{D}_{\text{test}}$.
- Given an input x , we define a feature vector $\phi(x) \in \mathbb{R}^d$, which maps x into a high-dimensional point, each feature representing some useful aspect of x .
- This demo implements the following learning algorithms: rote learning, nearest neighbors, and stochastic gradient descent for linear predictors.
- Rote learning computes, for each possible feature vector $\phi(x)$, a histogram of possible outputs y based on the training data. Given a new input x , the most frequent y for $\phi(x)$ is returned.
- Nearest neighbors returns a predictor, which given an input x , returns the output of the closest x' (that is, $\|\phi(x') - \phi(x)\|_2$ is the smallest).

Learning algorithms: loss minimization

- Now we discuss the **loss minimization framework**, which is to find \mathbf{w} that minimizes $\sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w}) + \text{Penalty}(\mathbf{w})$. This captures many of the learning algorithms based on linear predictors.
- The possible loss functions for binary classification are $\text{Loss}_{\text{perceptron}}$ (Perceptron algorithm), $\text{Loss}_{\text{hinge}}$ (support vector machines), and $\text{Loss}_{\text{logistic}}$ (logistic regression). The possible loss functions for regression are $\text{Loss}_{\text{squared}}$ (least squares regression) and $\text{Loss}_{\text{absdev}}$ (least absolute deviations regression).
- We will only work with L_2 regularization, in which $\text{Penalty}(\mathbf{w}) = \frac{\lambda}{2} \|\mathbf{w}\|^2$.
- The linear predictor is specified by a weight vector $\mathbf{w} \in \mathbb{R}^d$, so that the prediction for regression is $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$ and $f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$ for binary classification.
- The stochastic gradient descent (SGD) algorithm is an iterative algorithm. It repeatedly picks up an example $(x, y) \in \mathcal{D}_{\text{train}}$ and updates the weights in a way to reduce $\text{Loss}(x, y, \mathbf{w})$. This is done by computing the gradient and performing $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$, where η_t is the step size at iteration t . For example, we can take $\eta_t = 1/t^\alpha$, where α is the reduction.

Documentation

This demo allows you to create your own examples, feature extractors, and step through various learning algorithms.

- Problem definition
 - `trainExample(x, y)`, `testExample(x, y)`: adds a training/test example with input `x` and output `y` (e.g., `trainExample('hello', +1)`).
 - `featureExtractor(func)`: adds a feature extractor `func(ex, fv)`, which takes an example object `ex = {x:x, y:y}` and a feature vector `fv`, which you can add features to by calling `fv.add(feature, value)`.
- Learning algorithms
 - `roteLearning(opts)`: simply memorizes the training data.
 - `nearestNeighbors(opts)`: run the nearest neighbors algorithm.
 - `sgd(opts)`: run stochastic gradient descent on the `opts.loss` (which can be perceptron, hinge, logistic, squared, absdev). `opts.lambda` is the amount of regularization. `opts.steps` is the number of steps to take at once. `opts.reduction` is the reduction for the step size. `opts.ordered`: if set to true, don't randomize order of training examples.