Lecture 3: Machine learning II

cs221.stanford.edu/q

Question

Can we obtain decision boundaries which are circles by using linear classifiers?

Yes
No

- The answer is yes. This might seem paradoxical since we are only working with linear classifiers. But as we will see later, **linear** refers to the relationship between the weight vector $\mathbf{w}$ and the prediction score (not the input x, which might not even be a real vector), whereas the decision boundary refers to how the prediction varies as a function of x.

Framework

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

Learner
Optimization problem
Optimization algorithm

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

Review: optimization problem

Key idea: minimize training loss

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

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

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

Review: loss functions

Regression

Binary classification

Captures properties of the desired predictor

- The actual loss function depends on what we're trying to accomplish. Generally, the loss function takes the score $\mathbf{w} \cdot \phi(x)$, compares it with the correct output y to form either the residual (for regression) or the margin (for classification).
- Regression losses are smallest when the residual is close to zero. Classification losses are smallest when the margin is large. Which loss function we choose depends on the desired properties. For example, the absolute deviation loss for regression is robust against outliers. The logistic loss for classification never relents in encouraging large margin.
- Note that we've been talking about the loss on a single example, and plotting it in 1D against the residual or the margin. Recall that what we're actually optimizing is the training loss, which sums over all data points. To help visualize the connection between a single loss plot and the more general picture, consider the simple example of linear regression on three data points: $([1,0],2)$, $([1,0],4)$, and $([0,1],-1)$, where $\phi(x) = x$.
- Let's try to draw the training loss, which is a function of $\mathbf{w} = [w_1, w_2]$. Specifically, the training loss is $\frac{1}{3}((w_1-2)^2 + (w_1-4)^2 + (w_2-(-1))^2)$. The first two points contribute a quadratic term sensitive to w_1, and the third point contributes a quadratic term sensitive to w_2. When you combine them, you get a quadratic centered at $[3,-1]$. (Draw this on the whiteboard).

Review: optimization algorithms

Algorithm: gradient descent

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

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

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

Algorithm: stochastic gradient descent

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

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

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

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

- Finally, we introduced two very simple algorithms to minimize the training loss, both based on iteratively computing the gradient of the objective with respect to the parameters $\mathbf{w}$ and stepping in the opposite direction of the gradient. Think about a ball at the current weight vector and rolling it down on the surface of the training loss objective.
- Gradient descent (GD) computes the gradient of the full training loss, which can be slow for large datasets.
- Stochastic gradient descent (SGD), which approximates the gradient of the training loss with the loss at a single example, generally takes less time.
- In both cases, one must be careful to set the step size η properly (not too big, not too small).

Roadmap

Features

Neural networks

Gradients without tears

Nearest neighbors

- The first half of this lecture is about thinking about the feature extractor ϕ. Features are a critical part of machine learning which often does not get as much attention as it deserves. Ideally, they would be given to us by a domain expert, and all we (as machine learning people) have to do is to stick them into our learning algorithm. While one can get considerable mileage out of doing this, the interface between general-purpose machine learning and domain knowledge is often nuanced, so to be successful, it pays to understand this interface.
- In the second half of this lecture, we return to learning, rip out the linear predictors that we had from before, and show how we can build more powerful neural network classifiers given the features that we extracted.

Two components

Score (drives prediction):

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

- Previous: learning sets $\mathbf{w}$ via optimization
- Next: **feature extraction** specifies $\phi(x)$ based on domain knowledge

- As a reminder, the prediction is driven by the score $\mathbf{w} \cdot \phi(x)$. In regression, we predict the score directly, and in binary classification, we predict the sign of the score.
- Both $\mathbf{w}$ and $\phi(x)$ play an important role in prediction. So far, we have fixed $\phi(x)$ and used learning to set $\mathbf{w}$. Now, we will explore how $\phi(x)$ affects the prediction.

Organization of features

Task: predict whether a string is an email address

abc@gmail.com $\xrightarrow[\text{arbitrary!}]{\textbf{feature extractor}}$

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

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

- How would we go about 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: which strings should we include? We need a more systematic way to go about this.

Feature templates

Definition: feature template (informal)

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

Input:

abc@gmail.com

Some feature templates:

- Length greater than ___
- Last three characters equals ___
- Contains character ___
- Pixel intensity of position ___, ___

- 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".)
- A feature template also allows us to define a set of related features (contains_@, contains_a, contains_b). This reduces the amount of burden on the feature engineer since we don't need to know which particular characters are useful, but only that existence of certain single characters is a useful cue to look at.
- We can write each feature template as a English description with a blank (___), which is to be filled in with an arbitrary string. Also note that feature templates are most natural for defining binary features, ones which take on value 1 (true) or 0 (false).
- Note that an isolated feature (fraction of alphanumeric characters) can be treated as a trivial feature template with no blanks to be filled.
- As another example, if x is a $k \times k$ image, then $\{\text{pixelIntensity}_{ij} : 1 \leq i, j \leq k\}$ is a feature template consisting of k^2 features, whose values are the pixel intensities at various positions of x.

Feature templates

Feature template: last three characters equals ___

abc@gmail.com →

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

- This is an example of one feature template mapping onto a group of m^3 features, where m (26 in this example) is the number of possible characters.

Sparsity in feature vectors

Feature template: last character equals ___

abc@gmail.com →

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

Inefficient to represent all the zeros...

- In general, a feature template corresponds to many features. It would be inefficient to represent all the features explicitly. Fortunately, the feature vectors are often **sparse**, meaning that most of the feature values are 0. It is common for all but one of the features to be 0. This is known as a **one-hot representation** of a discrete value such as a character.

Feature vector representations

fracOfAlpha	: 0.85
contains_a	: 0
...	
contains_@	: 1
...	

Array representation (good for dense features):

```
[0.85, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Map representation (good for sparse features):

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

- Let's now talk a bit more about implementation. There are two common ways to define features: using arrays or using maps.
- **Arrays** assume a fixed ordering of the features and represent the feature values as an array. This representation 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 array representation is more common.
- However, when we have sparsity (few nonzeros), it is typically more efficient to represent the feature vector as a **map** from strings to doubles rather than a fixed-size array of doubles. The features not in the map implicitly have a default value of zero. This sparse representation is very useful in natural language processing, and is what allows us to work effectively over trillions of features. In Python, one would define a feature vector $\phi(x)$ as $\{$`"endsWith_"+x[-3:]: 1`$\}$. Maps do incur extra overhead compared to arrays, and therefore maps are much slower when the features are not sparse.
- Finally, it is important to be clear when describing features. Saying "length" might mean that there is one feature whose value is the length of x or that there could be a feature template "length is equal to ___". These two encodings of the same information can have a drastic impact on prediction accuracy when using a linear predictor, as we'll see later.

Hypothesis class

Predictor:

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

Definition: hypothesis class

A **hypothesis class** is the set of possible predictors with a fixed $\phi(x)$ and varying $\mathbf{w}$:

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

- Having discussed how feature templates can be used to organize groups of feature and allow us to leverage sparsity, let us further study how features impact prediction.
- The key notion is that of a **hypothesis class**, which is the set of all possible predictors that you can get by varying the weight vector $\mathbf{w}$. Thus, the feature extractor ϕ specifies a hypothesis class $\mathcal{F}$. This allows us to take data and learning out of the picture.

Feature extraction + learning

- Feature extraction: set $\mathcal{F}$ based on domain knowledge

- Learning: set $f_{\mathbf{w}} \in \mathcal{F}$ based on data

- Stepping back, we can see the two stages more clearly. First, we perform feature extraction (given domain knowledge) to specify a hypothesis class $\mathcal{F}$. Second, we perform learning (given training data) to obtain a particular predictor $f_{\mathbf{w}} \in \mathcal{F}$.
- Note that 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** predictors which are good? It's okay and expected that $\mathcal{F}$ will contain a bunch of bad ones as well.
- Later, we'll see reasons for keeping the hypothesis class small (both for computational and statistical reasons), because we can't get the optimal $\mathbf{w}$ for any feature extractor ϕ we choose.

Example: beyond linear functions

Regression: $x \in \mathbb{R}, y \in \mathbb{R}$

Linear functions:

$$\phi(x) = x$$

$$\mathcal{F}_1 = \{x \mapsto w_1 x + w_2 x^2 : w_1 \in \mathbb{R}, w_2 = 0\}$$

Quadratic functions:

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

$$\mathcal{F}_2 = \{x \mapsto w_1 x + w_2 x^2 : w_1 \in \mathbb{R}, w_2 \in \mathbb{R}\}$$

[whiteboard]

- Given a fixed feature extractor ϕ, let us consider the space of all predictors $f_{\mathbf{w}}$ obtained by sweeping $\mathbf{w}$ over all possible values.
- If we use $\phi(x) = x$, then we get linear functions that go through the origin.
- However, we want to have functions that "bend" (or are not monotonic). For example, if we want to predict someone's health given his or her body temperature. There is sweet spot temperature (37 C) where the health is optimal; both higher and lower values should cause the health to decline.
- If we use $\phi(x) = [x, x^2]$, then we get quadratic functions that go through the origin, which are a strict superset of the linear functions, and therefore are strictly more expressive.

Example: even more flexible functions

Regression: $x \in \mathbb{R}, y \in \mathbb{R}$

Piecewise constant functions:

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

$$\mathcal{F}_3 = \{x \mapsto \textstyle\sum_{j=1}^{10} w_j \mathbf{1}[j-1 < x \leq j] : \mathbf{w} \in \mathbb{R}^{10}\}$$

[whiteboard]

- However, even quadratic functions can be limiting, because they have to rise and fall in a certain (parabolic) way. What if we wanted a more flexible, freestyle approach?
- We can create piecewise constant functions by defining features that "fire" (are 1) on particular regions of the input (e.g., $1 < x \leq 2$). Each feature gets associated with its own weight, which in this case corresponds to the desired function value in that region.
- Thus by varying the weight vector, we get piecewise constant functions with a particular discretization level. We can increase or decrease the discretization level as we need.
- Advanced: what happens if x were not a scalar, but a d-dimensional vector? We could perform discretization in $\mathbb{R}^d$, but the number of features grows exponentially in d, which leads to a phenomenon called the curse of dimensionality.

Linear in what?

Prediction driven by score:

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

Linear in $\mathbf{w}$?	Yes
Linear in $\phi(x)$?	Yes
Linear in x?	No! (x not necessarily even a vector)

Key idea: non-linearity

- Predictors $f_{\mathbf{w}}(x)$ can be expressive **non-linear** functions and decision boundaries of x.
- Score $\mathbf{w} \cdot \phi(x)$ is **linear** function of $\mathbf{w}$, which permits efficient learning.

- Wait a minute...how were we able to get non-linear predictions using linear predictors?
- It is important to remember that for linear predictors, it is the score $\mathbf{w} \cdot \phi(x)$ that is linear in $\mathbf{w}$ and $\phi(x)$ (read it off directly from the formula). In particular, the score is not linear in x (it sometimes doesn't even make sense because x need not be a vector at all — it could be a string or a PDF file. Also, neither the predictor $f_{\mathbf{w}}$ (unless we're doing linear regression) nor the loss function TrainLoss($\mathbf{w}$) are linear in anything.
- The significance is as follows: From the feature extraction viewpoint, we can define arbitrary features that yield very **non-linear** functions in x. From the learning viewpoint (only looking at $\phi(x)$, not x), **linearity** plays an important role in being able to optimize the weights efficiently (as it leads to convex optimization problems).

Geometric viewpoint

$$\phi(x) = [1, x_1, x_2, x_1^2 + x_2^2]$$

How to relate **non-linear** decision boundary in x space with **linear** decision boundary in $\phi(x)$ space?

[demo]

- Let's try to understand the relationship between the non-linearity in x and linearity in $\phi(x)$. We consider binary classification where our input is $x = [x_1, x_2] \in \mathbb{R}^2$ a point on the plane. With the quadratic features $\phi(x)$, we can carve out the decision boundary corresponding to an ellipse (think about the formula for an ellipse and break it down into monomials).
- We can now look at the feature vectors $\phi(x)$, which include an extra dimension. In this 3D space, a linear predictor (defined by the hyperplane) actually corresponds to the non-linear predictor in the original 2D space.

An example task

Example: detecting responses

Input x:

two consecutive messages in a chat

Output $y \in \{+1, -1\}$:

whether the second message is a response to the first

Recall: feature extractor ϕ should pick out properties of x that might be useful for prediction of y

Recall: feature extractor ϕ returns a set of (feature name, real number) pairs

- Let's apply what you've learned about feature extraction to a concrete problem. The motivation here is that messaging platforms often just show a single stream of messages, when there is generally a grouping of messages into coherent conversations. How can we build a classifier that can group messages automatically? We can formulate this as a binary classification problem where we look at two messages and determine whether these two are part of the same conversation or not.

cs221.stanford.edu/q

Question

What feature templates would you use for predicting whether the second message is a response to the first?

time elapsed
time elapsed is between ___ and ___
first message contains ___
second message contains ___
two messages both contain ___
two messages have ___ common words

Summary so far

- Feature templates: organize related (sparse) features
- Hypothesis class: defined by features (what is possible)
- Linear classifiers: can produce non-linear decision boundaries

Roadmap

Features

Neural networks

Gradients without tears

Nearest neighbors

- What we've shown so far is that by being mildly clever with choosing the feature extractor ϕ, we can actually get quite a bit of mileage out of our so-called linear predictors.
- However, sometimes we don't know what features are good to use, either because the prediction task is non-intuitive or we don't have time to figure out which features are suitable. Sometimes, we think we might know what features are good, but then it turns out that they aren't (this happens a lot!).
- In the spirit of machine learning, we'd like to automate things as much as possible. In this context, it means creating algorithms that can take whatever crude features we have and turn them into refined predictions, thereby shifting the burden off feature extraction and moving it to learning.
- Neural networks have been around for many decades, but they fell out of favor because they were difficult to train. In the last decade, there has been a huge resurgence of interest in neural networks since they perform so well and training seems to not be such an issue when you have tons of data and compute.
- In a way, neural networks allow one to automatically learn the features of a linear classifier which are geared towards the desired task, rather than specifying them all by hand.

Motivation

Example: predicting car collision

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

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

True function: safe if cars sufficiently far

$$y = \text{sign}(|x_1 - x_2| - 1)$$

Examples:

x	y
$[1, 3]$	$+1$
$[3, 1]$	$+1$
$[1, 0.5]$	-1

- As a motivating example, consider the problem of predicting whether two cars at positions x_1 and x_2 are going to collide. Suppose the true output is 1 (safe) whenever the cars are separated by a distance of at least 1. Clearly, this the decision is not linear.

Decomposing the problem

Test if car 1 is far right of car 2:

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

Test if car 2 is far right of car 1:

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

Safe if at least one is true:

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

x	h_1	h_2	y
$[1, 3]$	0	1	$+1$
$[3, 1]$	1	0	$+1$
$[1, 0.5]$	0	0	-1

- The intuition is to break up the problem into two subproblems, which test if car 1 (car 2) is to the far right.
- Given these two binary values h_1, h_2, we can declare safety if at least one of them is true.

Learning strategy

Define: $\phi(x) = [1, x_1, x_2]$

Intermediate hidden subproblems:

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

$$h_2 = \mathbf{1}[\mathbf{v}_2 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_2 = [-1, -1, +1]$$

Final prediction:

$$f_{\mathbf{V},\mathbf{w}}(x) = \text{sign}(w_1 h_1 + w_2 h_2) \qquad \mathbf{w} = [1, 1]$$

Key idea: joint learning

Goal: learn both hidden subproblems $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2)$ and combination weights $\mathbf{w} = [w_1, w_2]$

- Having written y in a specific way, let us try to generalize to a family of predictors (this seems to be a recurring theme).
- We can define $\mathbf{v}_1 = [-1, 1, -1]$ and $\mathbf{v}_2 = [-1, -1, 1]$ and $w_1 = w_2 = 1$ to accomplish this.
- At a high-level, we have defined two intermediate subproblems, that of predicting h_1 and h_2. These two values are hidden in the sense that they are not specified to be anything. They just need to be set in a way such that y is linearly predictable from them.

Gradients

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

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

[whiteboard]

Definition: logistic function

The logistic function maps $(-\infty, \infty)$ to $[0, 1]$:

$$\sigma(z) = (1 + e^{-z})^{-1}$$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Solution:

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

- If we try to train the weights $\mathbf{v}_1, \mathbf{v}_2, w_1, w_2$, we will immediately notice a problem: the gradient of h_1 with respect to $\mathbf{v}_1$ is always zero because of the hard thresholding function.
- Therefore, we define a function **logistic function** $\sigma(z)$, which looks roughly like the step function $\mathbf{1}[z \geq 0]$, but has non-zero gradients everywhere.
- One thing to bear in mind is that even though the gradients are non-zero, they can be quite small when $|z|$ is large. This is what makes optimizing neural networks hard.

Linear predictors

Linear predictor:

Output:

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

- Let's try to visualize the predictors.
- Recall that linear classifiers take the input $\phi(x) \in \mathbb{R}^d$ and directly take the dot product with the weight vector $\mathbf{w}$ to form the score, the basis for prediction in both binary classification and regression.

Neural networks

Neural network:

Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

- The idea in neural networks is to map an input $\phi(x) \in \mathbb{R}^d$ onto a hidden **intermediate representation** $\mathbf{h} \in \mathbb{R}^k$, which in turn is mapped to the score.
- Specifically, let k be the number of hidden units. For each hidden unit $j = 1, \ldots, k$, we have a weight vector $\mathbf{v}_j \in \mathbb{R}^d$, which is used to determine the value of the hidden node $h_j \in \mathbb{R}$ (also called the **activation**) according to $h_j = \sigma(\mathbf{v}_j \cdot \phi(x))$, where σ is the activation function. The activation function can be a number of different things, but its main property is that it is a non-linear function. Traditionally the **sigmoid** function $\sigma(z) = (1 + e^{-z})^{-1}$ was used, but recently the **rectified linear** function $\sigma(z) = \max\{z, 0\}$ has gained popularity.
- Let $\mathbf{h} = [h_1, \ldots, h_k]$ be the vector of activations. This activation vector is now combined with another weight vector $\mathbf{w} \in \mathbb{R}^k$ to produce the final score.

Neural networks

Interpretation: intermediate hidden units as learned features of a linear predictor

Key idea: feature learning

Before: apply linear predictor on manually specify features

$$\phi(x)$$

Now: apply linear predictor on automatically learned features

$$h(x) = [h_1(x), \ldots, h_k(x)]$$

- The noteworthy aspect here is that the activation vector $\mathbf{h}$ behaves a lot like our feature vector $\phi(x)$ that we were using for linear prediction. The difference is that mapping from input $\phi(x)$ to $\mathbf{h}$ is learned automatically, not manually constructed (as was the case before). Therefore, a neural network can be viewed as learning the features of a linear classifier. Of course, the type of features that can be learned must be of the form $x \to \sigma(\mathbf{v}_j \cdot \phi(x))$.
- Whether this is a suitable form depends on the nature of the application. Empirically, though, neural networks have been quite successful, since learning the features from the data with the explicit objective of minimizing the loss can yield better features than ones which are manually crafted. Recently, there have been some advances in getting neural networks to work, and they have become the state-of-the-art in many tasks. For example, all the major companies (Google, Microsoft, IBM) all recently switched over to using neural networks for speech recognition. In computer vision, (convolutional) neural networks are completely dominant in object recognition.

Roadmap

Features

Neural networks

Gradients without tears

Nearest neighbors

Motivation: loss minimization

Optimization problem:

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

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

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

$$f_{\mathbf{V},\mathbf{w}}(x) = \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x))$$

Goal: compute gradient

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

- The main thing left to do for neural networks is to be able to train them. Conceptually, this should be straightforward: just take the gradient and run SGD.
- While this is true, computing the gradient, even though it is not hard, can be quite tedious to do by hand.

Approach

Mathematically: just grind through the chain rule

Next: visualize the computation using a computation graph

Advantages:

- Avoid long equations
- Reveal structure of computations (modularity, efficiency, dependencies)

- We will illustrate a graphical way of organizing the computation of gradients, which is built out of a few components.
- This graphical approach will show the structure of the function and will not only make gradients easy to compute, but also shed more light onto the predictor and loss function.
- In fact, these days if you use a package such as TensorFlow or Pytorch, you can write down the expressions symbolically and the gradient is computed for you. This is done essentially using the computational procedure that we will see.

Functions as boxes

Partial derivatives (gradients): how much does the output change if an input changes?

Example:

$$2\text{in}_1 + (\text{in}_2 + \epsilon)\text{in}_3 = \text{out} + \text{in}_3\epsilon$$

- The first conceptual step is to think of functions as boxes that take a set of inputs and produces an output. Then the partial derivatives (gradients if the input is vector-valued) are just a measure of sensitivity: if we perturb in_1 by a small amount ϵ, how much does the output out change? The answer is $\frac{\partial \mathsf{out}}{\partial \mathsf{in}_1} \cdot \epsilon$. For convenience, we write the partial derivative on the edge connecting the input to the output.

Basic building blocks

- Here are 5 examples of simple functions and their partial derivatives. These should be familiar from basic calculus. All we've done is to present them in a visually more intuitive way.
- But 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 in machine learning.

Composing functions

Chain rule:

$$\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$$

- The second conceptual point is to think about **composing**. Graphically, this is very natural: the output of one function f simply gets fed as the input into another function g.
- Now how does in affect out (what is the partial derivative)? The key idea is that the partial derivative **decomposes** into a product of the two partial derivatives on the two edges. You should recognize this is no more than the chain rule in graphical form.
- More generally, if the partial derivative of y with respect to x is simply the product of all the green expressions on the edges of the path connecting x and y. This visual intuition will help us better understand more complex functions, which we will turn to next.

Binary classification with hinge loss

Hinge loss:

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

Compute:

$$\frac{\partial \text{Loss}(x, y, \mathbf{w})}{\partial \mathbf{w}}$$

Binary classification with hinge loss

Gradient: multiply the edges

$$-\mathbf{1}[\text{margin} < 1]\phi(x)y$$

- Let us start with a simple example: the hinge loss for binary classification.
- In red, we have highlighted the weights $\mathbf{w}$ with respect to which we want to take the derivative. The central question is how small perturbations in $\mathbf{w}$ affect a change in the output (loss). Intermediate nodes have been labeled with interpretable names (score, margin).
- The actual gradient is the product of the edge-wise gradients from $\mathbf{w}$ to the loss output.

Neural network

$$\text{Loss}(x, y, \mathbf{w}) = \left(\sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

- Now, we can apply the same strategy to neural networks. Here we are using the squared loss for concreteness, but one can also use the logistic or hinge losses.
- Note that there is some really nice modularity here: you can pick any predictor (linear or neural network) to drive the score, and the score can be fed into any loss function (squared, hinge, etc.).

Backpropagation

Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

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

- So far, we have mainly used the graphical representation to visualize the computation of function values and gradients for our conceptual understanding. But it turns out that the graph has algorithmic implications too.
- Recall that to train any sort of model using (stochastic) gradient descent, we need to compute the gradient of the loss (top output node) with respect to the weights (leaf nodes highlighted in red).
- We also saw that these gradients (partial derivatives) are just the product of the local derivatives (green stuff) along the path from a leaf to a root. So we can just go ahead and compute these gradients: for each red node, multiply the quantities on the edges. However, notice that many of the paths share subpaths in common, so sometimes there's an opportunity to save computation (think dynamic programming).
- To make this sharing more explicit, for each node i in the tree, define the forward value f_i to be the value of the subexpression rooted at that tree, which depends on the inputs underneath that subtree. For example, the parent node of w_1 corresponds to the expression $w_1\sigma(\mathbf{v}_1 \cdot \phi(x))$. The f_i's are the intermediate computations required to even evaluate the function at the root.
- Next, for each node i in the tree, define the backward value g_i to be the gradient of the output with respect to f_i, the forward value of node i. This measures the change that would happen in the output (root node) induced by changes to f_i.
- Note that both f_i and g_i can either be scalars, vectors, or matrices, but have the same dimensionality.

Backpropagation

Algorithm: backpropagation

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

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

- We now define the backpropagation algorithm on arbitrary computation graphs.
- First, in the forward pass, we go through all the nodes in the computation graph from leaves to the root, and compute f_i, the value of each node i, recursively given the node values of the children of i. These values will be used in the backward pass.
- Next, in the backward pass, we go through all the nodes from the root to the leaves and compute g_i recursively from f_i and g_j, the backward value for the parent of i using the key recurrence $g_i = \frac{\partial f_j}{\partial f_i} g_j$ (just the chain rule).
- In this example, the backward pass gives us the gradient of the output node (the gradient of the loss) with respect to the weights (the red nodes).

Roadmap

Features

Neural networks

Gradients without tears

Nearest neighbors

- Linear predictors were governed by a simple dot product $\mathbf{w} \cdot \phi(x)$. Neural networks chained together these simple primitives to yield something more complex. Now, we will consider **nearest neighbors**, which yields complexity by another mechanism: computing similarities between examples.

Nearest neighbors

Algorithm: nearest neighbors

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

Predictor $f(x')$:

- Find $(x, y) \in \mathcal{D}_{\text{train}}$ where $\|\phi(x) - \phi(x')\|$ is smallest
- Return y

Key idea: similarity

Similar examples tend to have similar outputs.

- **Nearest neighbors** is perhaps conceptually one of the simplest learning algorithms. In a way, there is no learning. At training time, we just store the entire training examples. At prediction time, we get an input x' and we just find the input in our training set that is **most similar**, and return its output.
- In a practical implementation, finding the closest input is non-trivial. Popular choices are using k-d trees or locality-sensitive hashing. We will not worry about this issue.
- The intuition being expressed here is that similar (nearby) points tend to have similar outputs. This is a reasonable assumption in most cases; all else equal, having a body temperature of 37 and 37.1 is probably not going to affect the health prediction by much.

Expressivity of nearest neighbors

Decision boundary: based on Voronoi diagram

- Much more expressive than quadratic features
- **Non-parametric**: the hypothesis class adapts to number of examples
- Simple and powerful, but kind of brute force

- Let's look at the decision boundary of nearest neighbors. The input space is partitioned into regions, such that each region has the same closest point (this is a Voronoi diagram), and each region could get a different output.
- Notice that this decision boundary is much more expressive than what you could get with quadratic features. In particular, one interesting property is that the complexity of the decision boundary adapts to the number of training examples. As we increase the number of training examples, the number of regions will also increase. Such methods are called **non-parametric**.

cs221.stanford.edu/q

Question

What was the most surprising thing you learned today?

Summary of learners

- Linear predictors: combine raw features

  prediction is **fast**, **easy** to learn, **weak** use of features

- Neural networks: combine learned features

  prediction is **fast**, **hard** to learn, **powerful** use of features

- Nearest neighbors: predict according to similar examples

  prediction is **slow**, **easy** to learn, **powerful** use of features

- Let us conclude now. First, we discussed some general principles for designing good features for linear predictors. Just with the machinery of linear prediction, we were able to obtain rich predictors which were quite rich.
- Second, we focused on expanding the expressivity of our predictors fixing a particular feature extractor ϕ.
- We covered three algorithms: **linear predictors** combine the features linearly (which is rather weak), but is easy and fast.
- **Neural networks** effectively learn non-linear features, which are then used in a linear way. This is what gives them their power and prediction speed, but they are harder to learn (due to the non-convexity of the objective function).
- **Nearest neighbors** is based on computing similarities with training examples. They are powerful and easy to learn, but are slow to use for prediction because they involve enumerating (or looking up points in) the training data.