Algorithms for Artificial Intelligence

Stanford, California

Contents

Acknowledgments v

1 *Machine Learning I* 1
1.1 *Linear Predictors* 1
1.1.1 *Types of Prediction Tasks* 1
1.1.2 *Data* 2
1.1.3 *Learning* 3
1.1.4 *Feature Extraction* 3
1.1.5 *Weight Vector* 5
1.1.6 *Binary Classification* 7
1.1.7 *Geometric Intuition* 7
1.2 *Loss Minimization* 8
1.2.1 *Score and Margin* 9
1.2.2 *Linear Regression* 10
1.2.3 *Regression Loss Functions* 11
1.2.4 *Loss Minimization Framework* 12
1.3 *Optimization Problem* 13
1.3.1 *Gradient Descent* 13
1.3.2 *Least Squares Regression* 14

1.3.3 *Gradient Descent is Slow* 15
1.3.4 *Stochastic Gradient Descent* 16
1.3.5 *Step Size* 17
1.3.6 *Summary* 17
1.3.7 *Zero-One Loss* 18
1.3.8 *Hinge Loss (SVMs)* 18
1.3.9 *Logistic Regression* 19
1.3.10 *Summary* 21

2 *Machine Learning II* 22
2.1 *Features* 25
2.1.1 *Organization of Features* 26
2.1.2 *Feature Templates* 26
2.1.3 *Sparsity in Feature Vectors* 27
2.1.4 *Feature Vector Representation* 27
2.1.5 *Hypothesis Class* 29
2.1.6 *Feature Extration and Learning* 29
2.1.7 *Linearity* 31
2.1.8 *Geometric viewpoint* 32
2.1.9 *Summary* 33
2.2 *Neural Networks* 33
2.2.1 *Decomposing the problem* 34
2.2.2 *Learning strategy* 35
2.2.3 *Gradients* 35
2.2.4 *Linear functions* 36
2.2.5 *Neural networks* 36

2.3 *Efficient Gradients* 38
2.3.1 *Approach* 39
2.3.2 *Functions as boxes* 39
2.3.3 *Basic building blocks* 40
2.3.4 *Composing functions* 40
2.3.5 *Binary classification with hinge loss* 41
2.3.6 *Backpropagation* 41
2.4 *Nearest Neighbors* 43
2.4.1 *Summary of learners* 45

3 *Markov Decision Process I* 46
3.1 *Uncertainty* 46
3.2 *Markov Decision Processes* 47
3.2.1 *Transitions* 50
3.2.2 *Solutions* 51
3.3 *Policy Evaluation* 52
3.3.1 *Discounting* 54
3.3.2 *Policy evaluation* 54
3.3.3 *Summary* 59
3.4 *Value Iteration* 59
3.4.1 *Optimal value and policy* 59
3.4.2 *Value iteration* 60
3.4.3 *Convergence* 60
3.4.4 *Summary of algorithms* 61
3.4.5 *Unifying idea* 61
3.4.6 *Summary* 63

References 64

Acknowledgments

This work is taken from the lecture notes for the course *Artificial Intelligence: Principles and Techniques* at Stanford University, CS 221 (`cs221.stanford.edu`). The contributors to the content of this work are Percy Liang and Dorsa Sadigh—this collection is simply a typesetting of existing lecture notes with minor modifications and additions of working Julia implementations. We would like to thank the original authors for their contribution. In addition, we wish to thank Mykel Kochenderfer and Tim Wheeler for their contribution to the Tufte-Algorithms LaTeX template, based off of *Algorithms for Optimization*.[1]

[1] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

Robert J. Moss
Stanford, Calif.
September 15, 2020

Ancillary material is available on the template's webpage:
`https://github.com/sisl/textbook_template`

1 Machine Learning I

From CS221 Spring 2020, Percy Liang, Chelsea Finn & Nima Anari, Stanford University.

1.1 Linear Predictors

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

Example application: spam classification.
Input: $x =$ email message
Output: $y = \{\text{spam}, \text{not-spam}\}$
Objective: obtain a *predictor* f:

$$x \longrightarrow \boxed{f} \longrightarrow y$$

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

1.1.1 Types of Prediction Tasks

In the context of classification tasks, f is called a *classifier* and y is called a *label* (sometimes *class*, *category*, or *tag*). The key distinction between binary classifica-

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

- **Binary classification:** (e.g., email $\implies$ spam/not spam)

$$x \longrightarrow \boxed{f} \longrightarrow y \in \{+1, -1\}$$

- **Regression:** (e.g., location, year $\implies$ housing price)

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

- **Multiclass classification:** y is a category

$$\text{picture} \longrightarrow \boxed{f} \longrightarrow \text{``cat''}$$

- **Ranking:** y is a permutation

$$\boxed{1}\boxed{2}\boxed{3}\boxed{4} \longrightarrow \boxed{f} \longrightarrow 2\,3\,4\,1$$

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

$$\text{``}\textit{la casa blu}\text{''} \longrightarrow \boxed{f} \longrightarrow \text{``}\textit{the blue house}\text{''}$$

1.1.2 *Data*

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

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

$$(x, y)$$

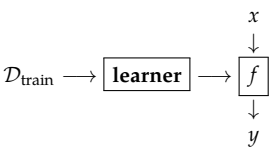
Training data: list of examples for spam classification

$$\mathcal{D}_{\text{train}} = \begin{bmatrix} (\text{“}\ldots\text{10m dollars}\ldots\text{”}, +1) \\ (\text{“}\ldots\text{CS221}\ldots\text{”}, -1) \end{bmatrix}$$

partial specification of behavior

1.1.3 Learning

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



1.1.4 Feature Extraction

We will consider predictors f based on *feature extractors* 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. It is okay to add features which turn out to be irrelevant, since the learning algorithm can sort it out (though it might require more data to do so).

Feature Vector Notation: Each input x represented by a *feature vector* $\phi(x)$, which is computed by the feature extractor ϕ. When designing features, it is useful to think of the feature vector as being a map from strings (feature names) to doubles (feature values). But formally, the feature vector $\phi(x) \in \mathbb{R}^d$ is a real

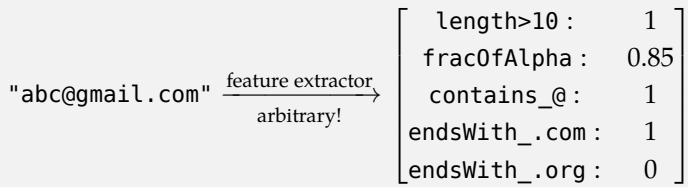
Example: feature extractor.

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

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

Feature extractor:

Given input x, output a set of (feature name, feature value) pairs.




```
# Indicator function \bbI
𝕀(b) = b ? 1 : 0

# Feature extrator \phi
ϕ = x -> [𝕀(length(x) > 10),
          sum(isletter(xᵢ) for xᵢ in x)/length(x),
          𝕀(occursin("@", x)),
          𝕀(endswith(x, ".com")),
          𝕀(endswith(x, ".org"))]

julia> x = "abc@gmail.com";
julia> ϕ(x)
5-element Array{Float64,1}:
 1.0
 0.8461538461538461
 1.0
 1.0
 0.0
```

Example 1.1. A *feature extractor* for classifying whether a string is an email address. The indicator function symbol 𝕀 can be created by typing `\bbI` and hitting tab and the feature extractor symbol ϕ symbol with `\phi`.

vector $\phi(x) = [\phi_1(x), \ldots, \phi_d(x)]$, where each component $\phi_j(x)$, for $j = 1, \ldots, d$, represents a feature.

Definition: feature vector. For an input x, its feature vector is:

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

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

This vector-based representation allows us to think about feature vectors as a point in a (high-dimensional) vector space, which will later be useful for getting some geometric intuition. Mathematically, feature vector doesn't need feature names:

$$\phi(x) = \begin{bmatrix} \texttt{length>10}: & 1 \\ \texttt{fracOfAlpha}: & 0.85 \\ \texttt{contains\_@}: & 1 \\ \texttt{endsWith\_.com}: & 1 \\ \texttt{endsWith\_.org}: & 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.85 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

1.1.5 *Weight Vector*

So far, we have defined a feature extractor ϕ that maps each input x to the feature vector $\phi(x)$. A *weight vector* $\mathbf{w} = [w_1, \ldots, w_d]$ (also called a *parameter vector* or *weights*) specifies the contributions of each feature vector to the prediction.

$$\mathbf{w} = \begin{bmatrix} \texttt{length>10}: & -1.2 \\ \texttt{fracOfAlpha}: & 0.6 \\ \texttt{contains\_@}: & 3 \\ \texttt{endsWith\_.com}: & 2.2 \\ \texttt{endsWith\_.org}: & 1.4 \end{bmatrix} = \begin{bmatrix} -1.2 \\ 0.6 \\ 3 \\ 2.2 \\ 1.4 \end{bmatrix}$$

In the context of binary classification with binary features ($\phi_j(x) \in \{0, 1\}$), the weights $w_j \in \mathbb{R}$ have an intuitive interpretation. If w_j is positive, then the presence of feature j ($\phi_j(x) = 1$) favors a positive classification. Conversely, if w_j is negative, then the presence of feature j favors a negative classification.

Definition: score. The score on an example (x, y) is $\mathbf{w} \cdot \phi(x)$, how *confident* we are in predicting $+1$. Score is a weighted combination of features:

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

```
score(x, w, ϕ) = w⋅ϕ(x)
```

Algorithm 1.1. The *score* of input `x` using weights **w** and feature extractor `ϕ` written in the Julia programming language. The `dot` product symbol ⋅ can be created by typing `\cdot` and hitting tab (included in the `LinearAlgebra` package), and the **w** symbol with `\bfw`.

Note that while the feature vector depends on the input x, the weight vector does not. This is because we want a single predictor (specified by the weight vector) that works on any input.

Given a feature vector $\phi(x)$ and a weight vector $\mathbf{w}$, we define the prediction *score* to be their inner product. The score intuitively represents the degree to which the classification is positive or negative. The predictor is linear because the score is a linear function of $\mathbf{w}$ (more on linearity in the next chapter). Again, in the context of binary classification with binary features, the score aggregates the contribution of each feature, weighted appropriately. We can think of each feature present as voting on the classification.

Example: score.

weight vector: $\mathbf{w} \in \mathbb{R}^d$ feature vector: $\phi(x) \in \mathbb{R}^d$

$$\begin{bmatrix} \texttt{length>10}: & -1.2 \\ \texttt{fracOfAlpha}: & 0.6 \\ \texttt{contains\_@}: & 3 \\ \texttt{endsWith\_.com}: & 2.2 \\ \texttt{endsWith\_.org}: & 1.4 \end{bmatrix} \quad \begin{bmatrix} \texttt{length>10}: & 1 \\ \texttt{fracOfAlpha}: & 0.85 \\ \texttt{contains\_@}: & 1 \\ \texttt{endsWith\_.com}: & 1 \\ \texttt{endsWith\_.org}: & 0 \end{bmatrix}$$

$$\mathbf{w} \cdot \phi(x) = -1.2(1) + 0.6(0.85) + 3(1) + 2.2(1) + 1.4(0) = 4.51$$

```
julia> x = "abc@gmail.com";
julia> w = [-1.2, 0.6, 3, 2.2, 1.4];
julia> score(x, w, ϕ)
4.507692307692308
```

Example 1.2. Calculating *score* as the weighted sum of features with feature extractor `ϕ` from example 1.1.

1.1.6 Binary Classification

We now have gathered enough intuition that we can formally define the predictor f. For each weight vector $\mathbf{w}$, we write $f_{\mathbf{w}}$ to denote the predictor that depends on $\mathbf{w}$ and takes the sign of the score. For this section, we will focus on the case of *binary classification*. For weight vector $\mathbf{w} \in \mathbb{R}^d$ and feature vector $\phi(x) \in \mathbb{R}^d$, a (binary) linear classifier is defined as:

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \phi(x) > 0 \\ -1 & \text{if } \mathbf{w} \cdot \phi(x) < 0 \\ ? & \text{if } \mathbf{w} \cdot \phi(x) = 0 \end{cases}$$

Recall that in this setting, we call the predictor a (binary) classifier. The case of $f_{\mathbf{w}}(x) = ?$ is a boundary case that isn't so important. We can just predict $+1$ arbitrarily as a matter of convention.

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \phi(x) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

```
binary_classifier(x, w, ϕ) = score(x, w, ϕ) ≥ 0 ? +1 : -1
```

Algorithm 1.2. A *binary classifier* f to classify input `x` using weight vector `w` and feature extractor `ϕ`. We predict $+1$ when the score is zero as a matter of convention.

1.1.7 Geometric Intuition

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

Example:

$$\mathbf{w} = [2, -1]$$
$$\phi(x) \in \{[2,0], [0,2], [2,4]\}$$

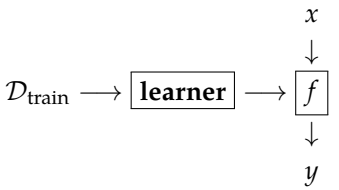
In general: binary classifier $f_{\mathbf{w}}$ defines a hyperplane **decision boundary** with normal vector $\mathbf{w}$.

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

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

1.2 *Loss Minimization*

So far we have talked about linear predictors $f_{\mathbf{w}}$ which are based on a feature extractor ϕ and a weight vector $\mathbf{w}$. Now we turn to the problem of estimating (also known as *fitting* or *learning*) $\mathbf{w}$ from training data. The *loss minimization* framework is to cast learning as an optimization problem. Note the theme of separating your problem into a model (optimization problem) and an algorithm (optimization algorithm).



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

1.2.1 *Score and Margin*

Recal that the *score* on an example (x, y) is $\mathbf{w} \cdot \phi(x)$ which is how *confident* we are in predicting $+1$. Before we talk about what loss functions look like and how to learn $\mathbf{w}$, we introduce another important concept, the notion of a *margin*. Suppose the correct label is $y \in \{-1, +1\}$. The margin of an input x is $\mathbf{w} \cdot \phi(x)y$, which measures how correct the prediction that $\mathbf{w}$ makes is. The larger the margin the better, and non-positive margins correspond to classification errors.

Definition: margin. The margin on an example (x, y) is how *correct* we are:

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

```
margin(x, y, w, ϕ) = score(x, w, ϕ)*y
```

Algorithm 1.3. The *margin* of an example `(x,y)` using weights `w` and feature extractor `ϕ`.

Note that if we look at the actual prediction $f_{\mathbf{w}}(x)$, we can only ascertain whether the prediction was right or not. By looking at the score and the margin, we can get a more nuanced view into the behavior of the classifier.

Geometrically, if $\|\mathbf{w}\| = 1$, then the margin of an input x is exactly the distance from its feature vector $\phi(x)$ to the *decision boundary*.

Question: When does a binary classifier err on an example?

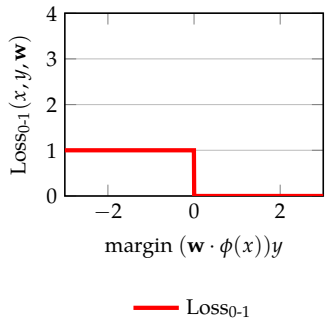
- margin less than 0
- margin greater than 0
- score less than 0
- score greater than 0

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

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

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

Figure 1.1. *Zero-one loss.*




```
Loss_01_f(x, y, w, ϕ, f) = 𝟙(f(x, w, ϕ) ≠ y)
Loss_01(x, y, w, ϕ) = 𝟙(margin(x, y, w, ϕ) ≤ 0)
```

Algorithm 1.4. The *zero-one loss* function for an example `(x,y)` using weights `w`, feature extractor `ϕ`, and classifier `f`. Note that `Loss_01` performs binary classification using the `margin`.

1.2.2 *Linear Regression*

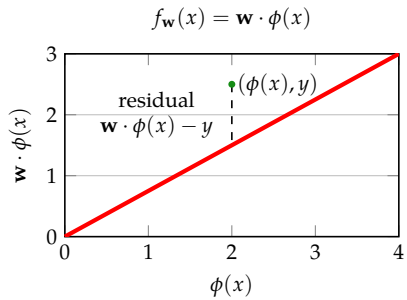
Now let's turn for a moment to regression, where the output y is a real number rather than $\{-1, +1\}$. Here, the *zero-one loss* doesn't make sense, because it's unlikely that we're going to predict y exactly.

Let's instead define the *residual* to measure how close the prediction $f_{\mathbf{w}}(x)$ is to the correct y. The residual will play the analogous role of the margin for classification and will let us craft an appropriate loss function.

Definition: residual. The *residual* is the amount by which the prediction $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$ overshoots the target y:

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

Figure 1.2. *Residual.*




```
residual(x, y, w, ϕ) = score(x, w, ϕ) - y
```

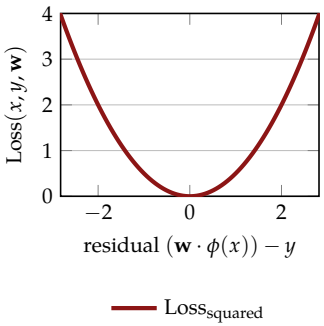
Algorithm 1.5. The *residual* of an example `(x,y)` using weights `w`, and feature extractor `ϕ`.

1.2.3 *Regression Loss Functions*

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

$$\begin{aligned}\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) &= (f_{\mathbf{w}}(x) - y)^2 \\ &= (\underbrace{\mathbf{w} \cdot \phi(x) - y}_{\text{residual}})^2\end{aligned}$$

```
Loss_squared(x, y, w, ϕ) = residual(x, y, w, ϕ)^2
```

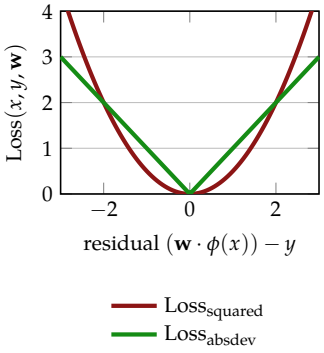


Algorithm 1.6. The *squared loss* for an example `(x,y)` using weights `w`, and feature extractor `ϕ`.

An alternative to the squared loss is the *absolute deviation loss*, which simply takes the absolute value of the residual.

$$\begin{aligned}\text{Loss}_{\text{absdev}}(x, y, \mathbf{w}) &= |f_{\mathbf{w}}(x) - y| \\ &= |\underbrace{\mathbf{w} \cdot \phi(x) - y}_{\text{residual}}|\end{aligned}$$

```
Loss_absdev(x, y, w, ϕ) = abs(residual(x, y, w, ϕ))
```



Algorithm 1.7. The *absolute deviation loss* for an example `(x,y)` using weights `w`, and feature extractor `ϕ`.

Example of $\text{Loss}_{\text{squared}}$ and $\text{Loss}_{\text{absdev}}$:

$$\mathbf{w} = [2, -1], \phi(x) = [2, 0], y = -1$$
$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = 25$$
$$\text{Loss}_{\text{absdev}}(x, y, \mathbf{w}) = 5$$

```
julia> w = [2, -1]; ϕ = x->[2, 0]; y = -1; x = missing
julia> Loss_squared(x, y, w, ϕ)
25
julia> Loss_absdev(x, y, w, ϕ)
5
```

1.2.4 *Loss Minimization Framework*

So far: one example, $\text{Loss}(x, y, \mathbf{w})$ is easy to minimize. Note that on one example, both the squared and absolute deviation loss functions have the same minimum, so we cannot really appreciate the differences here. However, we are learning $\mathbf{w}$ based on a whole training set $\mathcal{D}_{\text{train}}$, not just one example. We typically minimize the *training loss* (also known as the training error or empirical risk), which is the average loss over all the training examples.

Key idea: minimize training loss

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

$$\underset{\mathbf{w} \in \mathbb{R}^d}{\text{minimize}}\, \text{TrainLoss}(\mathbf{w})$$

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

```
function TrainLoss(w, 𝒟train, ϕ, Loss)
    return sum(Loss(x,y,w,ϕ) for (x,y) ∈ 𝒟train)/length(𝒟train)
end

minimize(𝒟train, ϕ, Loss) =
    minimize(w->TrainLoss(w, 𝒟train, ϕ, Loss))
```

Algorithm 1.8. *Training loss* for weights $\mathbf{w}$ and data `𝒟train` using feature extractor `ϕ` and loss function `Loss`.

Importantly, such an optimization problem requires making tradeoffs across all the examples (in general, we won't be able to set $\mathbf{w}$ to a single value that makes every example have low loss).

Now the question of which loss we should use becomes more interesting. For example, consider the case where all the inputs are $\phi(x) = 1$. Essentially the problem becomes one of predicting a single value y^* which is the least offensive towards all the examples.

If our loss function is the squared loss, then the optimal value is the mean $y^* = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} y$. If our loss function is the absolute deviation loss, then the optimal value is the median. The median is more robust to outliers: you can

Which regression loss to use? For the training data $\mathcal{D}_{\text{train}}$ and feature vector $\phi(x) = x$, the weight $\mathbf{w}$ that minimizes the *least squares* (L_2) regression training loss is the **mean** y. The mean tries to accommodate every example, i.e. popular.

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

For *least absolute deviation* (L_1) regression, the weights $\mathbf{w}$ that minimizes the training loss is the **median** y. The median is more robust to outliers.

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

move the furthest point arbitrarily farther out without affecting the median. This makes sense given that the squared loss penalizes large residuals a lot more.

In summary, this is an example of where the choice of the loss function has a qualitative impact on the weights learned, and we can study these differences in terms of the objective function without thinking about optimization algorithms.

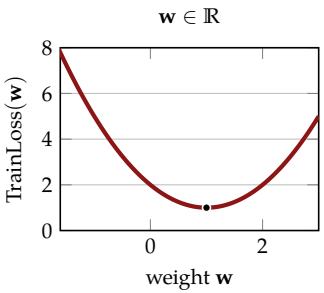


Figure 1.3. *Training loss* in $\mathbb{R}$ with minimum.

1.3 Optimization Problem

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

$$\underset{\mathbf{w} \in \mathbb{R}^d}{\text{minimize}} \, \text{TrainLoss}(\mathbf{w})$$

1.3.1 Gradient Descent

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

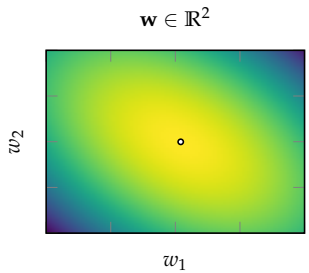


Figure 1.4. *Training loss* in $\mathbb{R}^2$ with minimum.

A general approach is to use *iterative optimization*, which essentially starts at some starting point $\mathbf{w}$ (say, all zeros), and tries to tweak $\mathbf{w}$ so that the objective function value decreases.

function GRADIENTDESCENT($\mathcal{D}_{\text{train}}$, ∇ TrainLoss)
  Initialize $\mathbf{w} = [0, \ldots, 0]$
  for $t = 1, \ldots, T$ **do**:
    $\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$

Algorithm 1.9. Gradient descent.

```
init_weights(d) = d == 1 ? zero(d) : zeros(d)

function gradient_descent(𝒟train, ϕ, ∇TrainLoss; η=0.1, T=100)
    w = init_weights(length(ϕ(𝒟train[1][1])))
    for t in 1:T
        w = w - η*∇TrainLoss(w, 𝒟train, ϕ)
    end
    return w
end
```

Algorithm 1.10. *Gradient descent* on the weights $\mathbf{w}$ to minimize the *least squares* training loss function over the training data `𝒟train` using feature extractor `ϕ`, step size `η`, and number of iterations `T`.

To do this, we will rely on the gradient of the function, which tells us which direction to move in to decrease the objective the most. The gradient is a valuable piece of information, especially since we will often be optimizing in high dimensions (d on the order of thousands).

This iterative optimization procedure is called *gradient descent*. Gradient descent has two *hyperparameters*, the *step size* η (which specifies how aggressively we want to pursue a direction) and the number of iterations T. Let's not worry about how to set them, but you can think of $T = 100$ and $\eta = 0.1$ for now.

1.3.2 *Least Squares Regression*

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

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

Compute the gradient using the chain rule:

$$\begin{aligned}\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) &= \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \nabla_{\mathbf{w}} \text{Loss}_{\text{squared}}(x, y, \mathbf{w}) \\ &= \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} 2(\underbrace{\mathbf{w}\cdot\phi(x) - y}_{\substack{\text{prediction}-\text{target} \\ \text{(residual)}}})\phi(x)\end{aligned}$$

Note that for linear predictors, the gradient is always something times $\phi(x)$ because $\mathbf{w}$ only affects the loss through $\mathbf{w}\cdot\phi(x)$.

```
∇Loss_squared(x, y, w, ϕ) = 2residual(x, y, w, ϕ)*ϕ(x)

function ∇TrainLoss(w, 𝒟train, ϕ)
    return sum(∇Loss_squared(x, y, w, ϕ)
        for (x,y) ∈ 𝒟train)/length(𝒟train)
end
```

Algorithm 1.11. *Training loss gradient* for *least squares regression*.

1.3.3 *Gradient Descent is Slow*

The problem with gradient descent is each iteration requires going over all training examples—which is expensive when you have lots of data!

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

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

1.3.4 *Stochastic Gradient Descent*

In *stochastic gradient descent* (SGD), rather than looping through all the training examples to compute a single gradient and making one step, SGD loops through the examples (x, y) and updates the weights $\mathbf{w}$ based on *each* example. Each update is not as good because we're only looking at one example rather than all the examples, but we can make many more updates this way.

function StochasticGradientDescent($\mathcal{D}_{\text{train}}$, ∇ Loss)
  Initialize $\mathbf{w} = [0, \ldots, 0]$
  for $t = 1, \ldots, T$ **do**:
    for $(x, y) \in \mathcal{D}_{\text{train}}$ **do**:
      $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$

Key idea: stochastic updates. It's not about *quality*, it's about *quantity*.

Algorithm 1.12. Stochastic gradient descent.

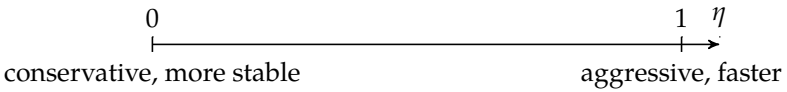
```
function stochastic_gradient_descent(𝒟train, ϕ, ∇Loss; η=0.1)
    w = init_weights(length(ϕ(𝒟train[1][1])))
    for t in 1:T
        for (x,y) ∈ 𝒟train
            w = w - η*∇Loss(x, y, w, ϕ)
        end
    end
    return w
end
```

Algorithm 1.13. *Stochastic gradient descent* over training data `𝒟train` using feature extractor `ϕ` and gradient of the loss function `∇Loss` and step size `η`.

In practice, we often find that just performing one pass over the training examples with SGD, touching each example once, often performs comparably to taking ten passes over the data with GD. There are also other variants of SGD. You can randomize the order in which you loop over the training data in each iteration, which is useful. Think about what would happen if you had all the positive examples first and the negative examples after that.

1.3.5 *Step Size*

One remaining issue is choosing the step size, which in practice (and as we have seen) is actually quite important. Generally, larger step sizes are like driving fast. You can get faster convergence, but you might also get very unstable results and crash and burn. On the other hand, with smaller step sizes you get more stability, but you might get to your destination more slowly. What should the *step size* η be?



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

A suggested form for the step size is to set the initial step size to 1 and let the step size decrease as the inverse of the square root of the number of updates we've taken so far. There are some nice theoretical results showing that SGD is guaranteed to converge in this case.[1]

[1] provided all your gradients have bounded length

```
mutable struct Decay
    i # iteration
end

import Base.*
*(δη::Decay, x) = 1/sqrt(δη.i+=1) * x
```

Algorithm 1.14. Inverse square root *step size decay*. The iteration `i` will automatically increment every time the `Decay` is multiplied.

1.3.6 *Summary*

In summary we have seen (1) the functions we're considering (linear predictors), (2) the criterion for choosing one (loss minimization), and (3) an algorithm that goes after that criterion (SGD). We already worked out a linear regression example. What are good loss functions for binary classification?

1. **Linear predictors**:

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

2. **Loss minimization**: learning as optimization

$$\underset{\mathbf{w}}{\text{minimize}}\, \text{TrainLoss}(\mathbf{w})$$

3. **Stochastic gradient descent**: optimization algorithm

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

1.3.7 *Zero-One Loss*

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

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

```
Loss_01(x, y, w, ϕ) = 𝕀(margin(x, y, w, ϕ) ≤ 0)
```

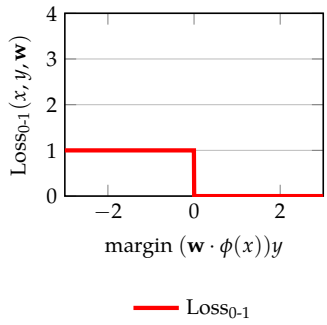
Problems with zero-one loss:

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

1.3.8 *Hinge Loss (SVMs)*

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

Figure 1.5. *Zero-one loss.*



Hinge loss upper bounds $\text{Loss}_{0\text{-}1}$ and has a non-trivial gradient. Try to increase the margin if it is less than 1. To fix this problem, we can use the *hinge loss*, which is an upper bound on the zero-one loss. Minimizing upper bounds are a general idea; the hope is that pushing down the upper bound leads to pushing down the actual function.

```
Loss_hinge(x, y, w, ϕ) = max(1 - margin(x, y, w, ϕ), 0)
```

Algorithm 1.15. The *hinge loss* function, an upper bound on the *zero-one* loss function.

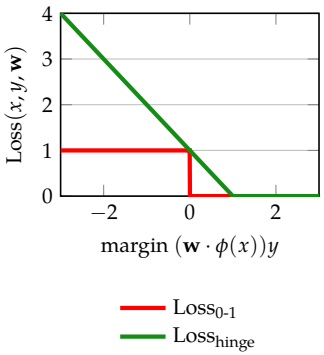


Figure 1.6. *Hinge loss.*

Advanced: The hinge loss corresponds to the *Support Vector Machine* (SVM) objective function with one important difference. The SVM objective function also includes a *regularization penalty* $\|\mathbf{w}\|^2$, which prevents the weights from getting too large. We will get to regularization later in the course, so you needn't worry about this for now. But if you're curious, read on.

Why should we penalize $\|\mathbf{w}\|^2$? One answer is Occam's razor, which says to find the simplest hypothesis that explains the data. Here, simplicity is measured in the length of $\mathbf{w}$. This can be made formal using statistical learning theory (take CS229T if you want to learn more).

Perhaps a less abstract and more geometric reason is the following. Recall that we defined the (algebraic) margin to be $\mathbf{w} \cdot \phi(x)y$. The actual (signed) distance from a point to the decision boundary is actually $\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \phi(x)y$, this is called the geometric margin. So the loss being zero (that is, $\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = 0$) is equivalent to the algebraic margin being at least 1 (that is, $\mathbf{w} \cdot \phi(x)y \geq 1$), which is equivalent to the geometric margin being larger than $\frac{1}{\|\mathbf{w}\|}$ (that is, $\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \phi(x)y \geq \frac{1}{\|\mathbf{w}\|}$). Therefore, reducing $\|\mathbf{w}\|$ increases the geometric margin. For this reason, SVMs are also referred to as *max-margin* classifiers.

```
geometric_margin(x, y, w, ϕ) = w/norm(w)⋅ϕ(x)*y
```

Algorithm 1.16. The *geometric margin* is the signed distance from a point to a decision boundary.

1.3.9 *Logistic Regression*

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

```
Loss_logistic(x, y, w, ϕ) = log(1 + exp(-margin(x, y, w, ϕ)))
```

Algorithm 1.17. The *logistic loss* function.

Try to increase margin even when it already exceeds 1. Another popular loss function used in machine learning is the *logistic loss*.

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

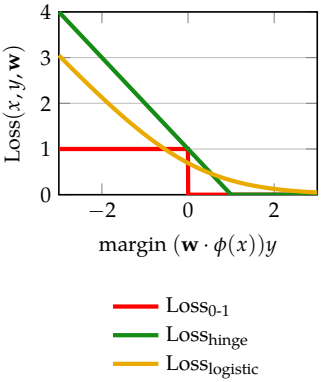


Figure 1.7. *Logistic loss*.

A gradient exercise. Compute the gradient of the hinge loss:

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

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.

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

What about when the margin is exactly 1? Technically, the gradient doesn't exist because the hinge loss is not differentiable there. Practically speaking, we can take either $-\phi(x)y$ or 0 (or anything in between).

```
∇Loss_hinge(x, y, w, ϕ) = margin(x, y, w, ϕ) < 1 ? -ϕ(x)*y : 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 $\mathbf{w}'$.

1.3.10 *Summary*

Table 1.1. Summary of machine learning.

score $= \mathbf{w} \cdot \phi(x)$	**Classification**	**Regression**
Predictor $f_{\mathbf{w}}$	sign(score)	score
Relate to correct y	margin (score $\cdot\, y$)	residual (score $- y$)
Loss functions	zero-one hinge logistic	squared absolute deviation
Algorithm	stochastic gradient descent	stochastic gradient descent

2 *Machine Learning II*

From CS221 Spring 2020, Percy Liang, Chelsea Finn & Nima Anari, Stanford University.

Recall from last chapter that learning is the process of taking training data and turning it into a model (predictor). Last chapter, 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. Recall that the optimization problem was to minimize the training loss, which is the average loss over all the training examples.

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

$$\underset{\mathbf{w} \in \mathbb{R}^d}{\text{minimize}}\, \text{TrainLoss}(\mathbf{w})$$

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.

Review: Optimization Algorithms. Finally, we introduced two very simple algorithms to minimize the training loss, both based on iteratively computing the

$\phi(x)$	y	$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$
$[1, 0]$	2	$(w_1 - 2)^2$
$[1, 0]$	4	$(w_1 - 4)^2$
$[0, 1]$	-1	$(w_1 - (-1))^2$

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{3}((w_1 - 2)^2 + (w_1 - 4)^2 + (w_2 - (-1))^2)$$

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 $[3, -1]$.

function GRADIENTDESCENT($\mathcal{D}_{\text{train}}$, ∇ TrainLoss)
  Initialize $\mathbf{w} = [0, \ldots, 0]$
  for $t = 1, \ldots, T$ **do**:
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

Algorithm 2.1. Gradient descent.

function STOCHASTICGRADIENTDESCENT($\mathcal{D}_{\text{train}}$, ∇ Loss)
  Initialize $\mathbf{w} = [0, \ldots, 0]$
  for $t = 1, \ldots, T$ **do**:
    for $(x, y) \in \mathcal{D}_{\text{train}}$ **do**:
      $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$

Algorithm 2.2. Stochastic gradient descent.

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).

The first half of this chapter is about thinking about the feature extractor ϕ. Features are a critical part of machine learning which often do not get as much attention as they deserve. 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 chapter, 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.

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

Advanced: Sometimes people might think that linear classifiers are not expressive, and that you need neural networks to get expressive and non-linear classifiers. This is false. You can build arbitrarily expressive models with the machinery of linear classifiers (see kernel methods). The advantages of neural networks are the computational benefits and the inductive bias that comes from the particular neural network architecture.

2.1 *Features*

There are two component to classification, score (drives prediction):

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

- Previous Chapter: **learning** chooses $\mathbf{w}$ via optimization.
- This Chapter: **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.

2.1.1 *Organization of Features*

Task: Predict whether a string is an email address.

$$\texttt{"abc@gmail.com"} \xrightarrow[\text{arbitrary!}]{\text{feature extractor}} \begin{bmatrix} \texttt{length>10}: & 1 \\ \texttt{fracOfAlpha}: & 0.85 \\ \texttt{contains\_@}: & 1 \\ \texttt{endsWith\_.com}: & 1 \\ \texttt{endsWith\_.org}: & 0 \end{bmatrix}$$

Which features to include? We 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.

2.1.2 *Feature Templates*

Definition: feature template (informal). A *feature template* is a group of features all computed in a similar way.

Input: `"abc@gmail.com"`, with some feature templates:

- Length greather 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 template example. Last three characters equals ___

$$\texttt{"abc@gmail.com"} \longrightarrow \begin{bmatrix} \texttt{endsWith\_aaa}: & 0 \\ \texttt{endsWith\_aab}: & 0 \\ & \cdots \\ \texttt{endsWith\_com}: & 1 \\ & \cdots \\ \texttt{endsWith\_zzz}: & 0 \end{bmatrix}$$

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.

```
last_three_chars___(x, Σ='a':'z') =
    [endswith(x, a*b*c) for a in Σ, b in Σ, c in Σ]
```

2.1.3 *Sparsity in Feature Vectors*

Innefficient 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.

2.1.4 *Feature Vector Representation*

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

Feature template example. Last character equals ___

$$\texttt{"abc@gmail.com"} \longrightarrow \begin{bmatrix} \texttt{endsWith\_a}: & 0 \\ \texttt{endsWith\_b}: & 0 \\ \texttt{endsWith\_c}: & 0 \\ & \dots \\ \texttt{endsWith\_z}: & 0 \end{bmatrix}$$

```
last_char_equals___(x, Σ='a':'z') = [endswith(x, c) for c in Σ]
```

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.

$$\begin{bmatrix} \texttt{fracOfAlpha}: & 0.85 \\ \texttt{contains\_a}: & 0 \\ & \dots \\ \texttt{endsWith\_@}: & 1 \\ & \dots \end{bmatrix}$$

- 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):

$$\{\texttt{"fracOfAlpha"}: 0.85, \texttt{"contains\_@"}: 1\}$$

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 the dictionary `"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.

2.1.5 Hypothesis Class

Having discussed how feature templates can be used to organize groups of features 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.

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

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

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

2.1.6 Feature Extration and Learning

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 $f_{\mathbf{w}} \in \mathcal{F}$.

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

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 2.1. Beyond linear functions.

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

Linear functions:

$$\phi(x) = x$$
$$\mathcal{F}_1 = \{x \mapsto w_1 x : w_2 \in \mathbb{R}\}$$

Quadratic functions:

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

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 a 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.

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 piece-

Example 2.2. Even more flexible functions

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

Piecewise constant functions:

$$\phi(x) = [\mathbb{1}[0 < x \leq 1], \mathbb{1}[1 < x \leq 2], \dots]$$
$$\mathcal{F}_3 = \left\{ x \mapsto \sum_{j=1}^{10} w_j \mathbb{1}[j-1 < x \leq j] : \mathbf{w} \in \mathbb{R}^{10} \right\}$$

wise 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.

2.1.7 *Linearity*

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 $\text{TrainLoss}(\mathbf{w})$ are linear in anything. For the 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.

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).

2.1.8 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? 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.

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.

Example 2.3. An example task: detecting responses.

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 or not these two are part of the same conversation.

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

2.1.9 *Summary*

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

2.2 *Neural Networks*

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 sense, 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.

Example 2.4. Predicting car collision.

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.
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

2.2.1 *Decomposing the problem*

The intuition is to break up the problem into two subproblems, which test if car 1 (or 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.

- Test if car 1 is far right of car 2: $h_1 = \mathbb{1}[x_1 - x_2 \geq 1]$
- Test if car 2 is far right of car 1: $h_2 = \mathbb{1}[x_2 - x_1 \geq 1]$
- 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

2.2.2 *Learning strategy*

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. Define: $\phi(x) = [1, x_1, x_2]$. Intermediate hidden subproblems:

$$\begin{aligned} h_1 &= \mathbb{1}[x_1 - x_2 \geq 1] \\ &= \mathbb{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_1 = [-1,+1,-1] \\ h_2 &= \mathbb{1}[x_2 - x_1 \geq 1] \\ &= \mathbb{1}[\mathbf{v}_2 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_2 = [-1,-1,+1] \end{aligned}$$

Final prediction:

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

Key idea: joint learning. Goal is to learn both hidden subproblems $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2)$ and combination weights $\mathbf{w} = [w_1, w_2]$.

2.2.3 *Gradients*

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 $\mathbb{1}[z \geq 0]$, 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.

Problem: gradient of h_1 with respect to $\mathbf{v}_1$ is 0: $h_1 = \mathbb{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$. Solution: $h_1 = \sigma(\mathbf{v}_1 \cdot \phi(x))$.

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

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

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \qquad \text{(derivative)}$$

```
σ(z) = 1/(1 + exp(-z))
σ´(z) = σ(z)*(1 - σ(z))
```

Algorithm 2.3. The logistic function σ and its derivative σ´.

2.2.4 *Linear functions*

Let's try to visualize the functions. Recall that a linear function takes 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. For linear functions, the output is the score $= \mathbf{w} \cdot \phi(x)$:

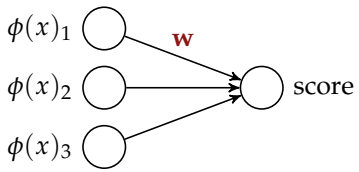


Figure 2.1. A linear function with $\phi(x) \in \mathbb{R}^3$.

2.2.5 *Neural networks*

A (one-layer) neural network first maps 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 via a linear function. 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. 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. The logistic function is an instance of an *activation function*, and is the classic one that was used in the past. These days, most people use a *rectifier* function, commonly known as a *rectified linear unit* (ReLU), which is defined as $\text{ReLU}(z) = \max(z, 0)$. The ReLU has two advantages: (i) its gradient doesn't vanish as z grows, which makes it empirically easier to train; and (ii) it only involves a max operation, which is computationally easier to compute than the exponential function.

```
ReLU(z) = max(z,0)
```

```
function neural_network(x, V, w, ϕ, g::Function=σ)
    h = map(vⱼ -> g(vⱼ ⋅ ϕ(x)), V)
    w ⋅ h
end

function neural_network(x, V, w, ϕ, g::Vector)
    h = map((g, vⱼ) -> g(vⱼ ⋅ ϕ(x)), g, V)
    w ⋅ h
end
```

Algorithm 2.4. A *one-layer neural network* with activation function `g` defaulting to the logistic function, or a vector of activation functions `g`.

For a neural network with one hidden layer, the intermediate hidden units are $h_j = \sigma(\mathbf{v}_j \cdot \phi(x))$ where $\sigma(z) = (1 + e^{-z})^{-1}$, producing output score $= \mathbf{w} \cdot \mathbf{h}$:

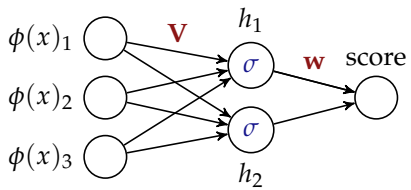


Figure 2.2. A *one-layer neural network* with $\phi(x) \in \mathbb{R}^3$ and $\mathbf{h} \in \mathbb{R}^2$ using the logistic activation function σ.

Interpretation: intermediate hidden units as learned features of a linear predictor. 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 $x \mapsto \sigma(\mathbf{v}_j \cdot \phi(x))$. Even for deep neural networks, no matter

now deep the neural network is, the top layer is always a linear function, and the layers below can be interpreted as defining a (possibly very complex) feature map. 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. Since 2010, 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.

Key idea: feature learning.

- **Before:** apply linear predictor on manually specified features.

$$\phi(x)$$

- **Now:** apply linear predictor on automatically learned features.

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

2.3 *Efficient Gradients*

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.

Optimization problem.

$$\begin{aligned}
\underset{\mathbf{V},\mathbf{w}}{\text{minimize}}\ & \text{TrainLoss}(\mathbf{V},\mathbf{w}) \\
\text{TrainLoss}(\mathbf{V},\mathbf{w}) &= \frac{1}{|\,\mathcal{D}_{\text{train}}\,|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x,y,\mathbf{V},\mathbf{w}) \\
\text{Loss}(x,y,\mathbf{V},\mathbf{w}) &= (f_{\mathbf{V},\mathbf{w}}(x)-y)^2 \\
f_{\mathbf{V},\mathbf{w}}(x) &= \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x))
\end{aligned}$$

Goal. compute the gradient: $\nabla_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V},\mathbf{w})$

2.3.1 *Approach*

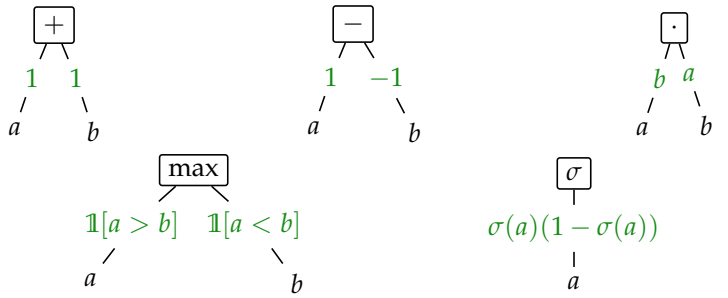
Mathematically, just grind through the chain rule. But wext we visualize the computation using a *computation graph*. We will illustrate a graphical way of organizing the computation of gradients, which is built out of a few components. This graphical approach will show the structure of the function 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.

Advantages:

- Avoid long equations
- Reveal structure of computations (modularity, efficiency, dependencies) — TensorFlow/PyTorch are built on this

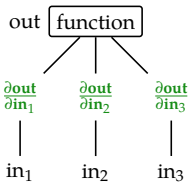
2.3.2 *Functions as boxes*

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 $\frac{\partial \text{out}}{\partial \text{in}_1} \cdot \epsilon$. For



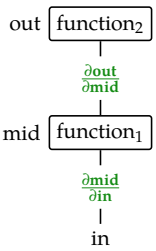
convenience, we write the partial derivative on the edge connecting the input to the output. Partial derivatives (i.e. gradients): how much does the output change if an input changes? Example:

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



2.3.3 *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 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.



2.3.4 *Composing functions*

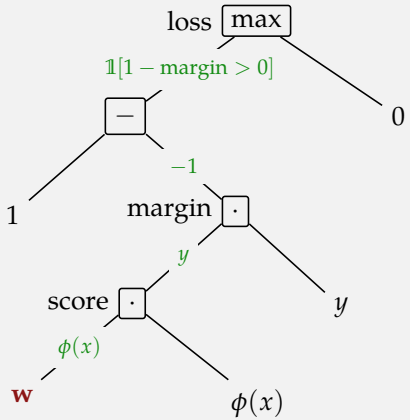
The second conceptual point is to think about *composing*. Graphically, this is very natural: the output of one function f simply gets fed as the input into another function g. Now how does **in** affect **out** (what is the partial derivative)? The key idea is that the partial derivative *decomposes* 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, 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.

Chain rule:

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

2.3.5 *Binary classification with hinge loss*

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.



Gradient with respect to $\mathbf{w}$. For the hinge loss

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

to compute

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

simply multiply the edges: $-\mathbb{1}[\text{margin} < 1]\phi(x)y$

2.3.6 *Backpropagation*

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.).

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

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.

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

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).

function BACKPROPAGATION(in)
  Forward pass, compute each f_i (from leaves to root)
  Backward pass, compute each g_i (from root to leaves)

Algorithm 2.5. Backpropagation.

While we can go through the motions of running the backpropagation algorithm to compute gradients, what is the result of running SGD? For linear predictors (using the squared loss or hinge loss), TrainLoss($\mathbf{w}$) is a convex function, which means that SGD (with an appropriately set step size) is theoretically guaranteed to converge to the global optimum. However, for neural networks, TrainLoss($\mathbf{w}$) 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 with proper attention to tuning hyperparameters. The gap between theory and practice is not well understood and an active area of research.

2.4 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 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*,[1] and return its output. In a practical implementation, finding the closest input is non-trivial. Popular choices are using

[1] Commonly used distances:

- Manhattan (L_1 norm):
$$\|\mathbf{v} - \mathbf{v}'\|_1 = \sum_{i=1}^{n} |v_i - v_i'|$$
- Euclidean (L_2 norm):
$$\|\mathbf{v} - \mathbf{v}'\|_2 = \sqrt{\sum_{i=1}^{n} (v_i - v_i')^2}$$

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.

function NEARESTNEIGHBORS($x', \phi, \mathcal{D}_{\text{train}}$)
  Training, just store $\mathcal{D}_{\text{train}}$
  for predictor $f(x')$ **do**
    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.

Algorithm 2.6. Nearest neighbors.

```
dist_manhattan(v, v′) = norm(v - v′, 1)
dist_euclidean(v, v′) = norm(v - v′, 2)

function nearest_neighbors(x′, ϕ, 𝒟train, dist)
    𝒟train[argmin([dist(ϕ(x), ϕ(x′)) for (x,y) in 𝒟train])][2]
end
```

Algorithm 2.7. *Nearest neighbors* to find the `y` value in the training data `𝒟train` associated with the example closets to the input `x′` after applying feature vector `ϕ`.

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*. The decision boundaries are shown in the Voronoi diagrams in figures 2.3 and 2.4.

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

Figure 2.3. *Voronoi diagram* showing *nearest neighbors* classification with *Manhattan* distance (L_1).

Figure 2.4. *Voronoi diagram* showing *nearest neighbors* classification with *Euclidean* distance (L_2).

2.4.1 *Summary of learners*

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. 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. Note that we can always make the hypothesis class arbitrarily large by adding more features, but that's another issue. *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.

1. **Linear predictors:** combine raw features
   - prediction is **fast**, **easy** to learn, **weak** use of features.
2. **Neural networks:** combine learned features
   - prediction is **fast**, **hard** to learn, **powerful** use of features.
3. **Nearest neighbors:** predict according to similar examples
   - prediction is **slow**, **easy** to learn, **powerful** use of features.

3 *Markov Decision Process I*

From CS221 Spring 2020, Percy Liang, Chelsea Finn & Nima Anari, Stanford University.

Last chapter, we looked at search problems, a powerful paradigm that can be used to solve a diverse range of problems ranging from word segmentation to package delivery to route finding. The key was to cast whatever problem we were interested in solving into the problem of finding the minimum cost path in a graph. However, search problems assume that taking an action a from a state s results *deterministically* in a unique successor state $\text{Succ}(s, a)$.

3.1 Uncertainty

In the real world, the deterministic successor assumption is often unrealistic, for there is *randomness*: taking an action might lead to any one of many possible states. One deep question here is how we can even hope to act optimally in the face of randomness? Certainly we can't just have a single deterministic plan, and talking about a minimum cost path doesn't make sense. Today, we will develop tools to tackle this more challenging setting. We will fortunately still be able to reuse many of the intuitions about search problems, in particular the notion of a state.

Applications. Randomness shows up in many places. They could be caused by limitations of the sensors and actuators of the robot (which we can control to some extent). Or they could be caused by market forces or nature, which we have no control over. We'll see that all of these sources of randomness can be handled in the same mathematical framework.

- Robotics: decide where to move, but actuators can fail, hit unseen obstacles, etc.

- Resource allocation: decide what to produce, don't know the customer demand for various products
- Agriculture: decide what to plant, but don't know weather and thus crop yield

Let us consider an example. You are exploring a South Pacific island, which is modeled as a 3×4 grid of states. From each state, you can take one of four actions to move to an adjacent state: north (N), east (E), south (S), or west (W). If you try to move off the grid, you remain in the same state. You start at $(2, 1)$. If you end up in either of the green or red squares, your journey ends, either in a lava lake (reward of -50) or in a safe area with either no view (2) or a fabulous view of the island (20). What do you do?

If we have a deterministic search problem, then the obvious thing will be to go for the fabulous view, which yields a reward of 20. You can set `numIters` to 10 and press `Run`. Each state is labeled with the maximum expected utility (sum of rewards) one can get from that state (analogue of FutureCost in a search problem). We will define this quantity formally later. For now, look at the arrows, which represent the best action to take from each cell. Note that in some cases, there is a tie for the best, where some of the actions seem to be moving in the wrong direction. This is because there is no penalty for moving around indefinitely. If you change `moveReward` to -0.1, then you'll see the arrows point in the right direction.

In reality, we are dealing with treacherous terrain, and there is on each action a probability `slipProb` of slipping, which results in moving in a random direction. Try setting `slipProb` to various values. For small values (e.g., 0.1), the optimal action is to still go for the fabulous view. For large values (e.g., 0.3), then it's better to go for the safe and boring 2. Play around with the other reward values to get intuition for the problem.

Important: note that we are only specifying the dynamics of the world, not directly specifying the best action to take. The best actions are computed automatically from the algorithms we'll see shortly.

3.2 *Markov Decision Processes*

Example: Dice game. For each round $r = 1, 2, \ldots$

- You choose **stay** or **quit**.
- If **quit**, you get \$10 and we end the game,
- If **stay**, you get \$4 and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.

Expected utility. If follow policy "stay":

$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \cdots = 12$$

Let's suppose you always stay. Note that each outcome of the game will result in a different sequence of rewards, resulting in a *utility*, which is in this case just the sum of the rewards. We are interested in the *expected* utility, which you can compute to be 12.

Expected utility. If follow policy "quit":

$$1(10) = 10$$

If you quit, then you'll get a reward of 10 deterministically. Therefore, in expectation, the "stay" strategy is preferred, even though sometimes you'll get less than 10.

While we already solved this game directly, we'd like to develop a more general framework for thinking about not just this game, but also other problems such as the volcano crossing example. To that end, let us formalize the dice game as a *Markov decision process* (MDP).

An MDP can be represented as a graph. The nodes in this graph include both *states* and *chance nodes*. Edges coming out of states are the possible actions from that state, which lead to chance nodes. Edges coming out of a chance nodes are the possible random outcomes of that action, which end up back in states. Our convention is to label these chance-to-state edges with the probability of a particular *transition* and the associated reward for traversing that edge.

Definition: Markov decision process.

- States: *the set of states*
- $s_{\text{start}} \in$ States: *starting state*
- Actions(s): *possible actions from state* s
- $T(s' \mid s, a)$: *probability of* s' *if take action* a *in state* s
- Reward(s, a, s'): *reward for the transition* (s, a, s')
- IsEnd(s): *whether at end of game*
- $0 \leq \gamma \leq 1$: *discount factor* (*default:* 1)

A *Markov decision process* has a set of states States, a starting state s_{state}, and the set of actions Actions(s) from each state s. It also has a *transition distribution* T, which specifies for each state s and action a, a distribution over possible successor states s'. Specifically, we have that $\sum_{s'} T(s, a, s') = 1$ because T is a probability distribution (more on this later). Associated with each transition (s, a, s') is a reward, which could be either positive or negative. If we arrive in a state s for which IsEnd(s) is true, then the game is over. Finally, the discount factor γ is a quantity which specifies how much we value the future and will be discussed later.

MDPs share many similarities with search problems, but there are differences (one main difference and one minor one). The main difference is the move from

a deterministic successor function $\text{Succ}(s, a)$ to transition probabilities over s'. We can think of the successor function $\text{Succ}(s, a)$ as a special case of transition probabilities:

$$T(s, a, s') = \begin{cases} 1 & \text{if } s' = \text{Succ}(s, a) \\ 0 & \text{otherwise} \end{cases}$$

A minor difference is that we've gone from minimizing costs to maximizing rewards. The two are really equivalent: you can negate one to get the other.

Definition: search problem.

- States: *the set of states*
- $s_{\text{state}} \in$ States: *starting state*
- $\text{Actions}(s)$: *possible actions from state s*
- $\text{Succ}(s, a)$: *where we end up if take action a in state s*
- $\text{Cost}(s, a)$: *cost for taking action a in state s*
- $\text{IsEnd}(s)$: *whether at end*

Map to MDP:

- $\text{Succ}(s, a) \Rightarrow T(s, a, s')$
- $\text{Cost}(s, a) \Rightarrow \text{Reward}(s, a, s')$

3.2.1 *Transitions*

Just to dwell on the major difference, transition probabilities, a bit more: for each state s and action a, the transition probabilities specifies a distribution over successor states s'.

Definition: transition probabilities. The *transition probabilities* $T(s, a, s')$ specify the probability of ending up in state s' if taken action a in state s.

Example: transition probabilities.

s	a	s'	$T(s,a,s')$
in	quit	end	1
in	stay	in	2/3
in	stay	end	1/3

Probabilities must sum to one. This means that for each given s and a, if we sum the transition probability $T(s,a,s')$ over all possible successor states s', we get 1. If a transition to a particular s' is not possible, then $T(s,a,s') = 0$. We refer to the s' for which $T(s,a,s') > 0$ as the successors. Generally, the number of successors of a given (s,a) is much smaller than the total number of states. For instance, in a search problem, each (s,a) has exactly one successor. Formally, for each state s and action a: $\sum_{s' \in \text{States}} T(s,a,s') = 1$ for successors $= s'$ such that $T(s,a,s') > 0$.

Let us revisit the transportation example. As we all know, magic trams aren't the most reliable forms of transportation, so let us asume that with probability $\frac{1}{2}$, it actually does as advertised, and with probability $\frac{1}{2}$ it just leaves you in the same state.

Example: transportation.

- Street with blocks numbered 1 to n.
- Walking from s to $s+1$ takes 1 minute.
- Taking a magic tram from s to $2s$ takes 2 minutes.
- How to travel from 1 to n in the least time?

Tram fails with probability 0.5.

3.2.2 *Solutions*

So we now know what an MDP is. What do we do with one? For search problems, we were trying to find the minimum cost *path*. However, fixed paths won't suffice

for MDPs, because we don't know which states the random dice rolls are going to take us. Therefore, we define a *policy*, which specifies an action for every single state, not just the states along a path. This way, we have all our bases covered, and know what action to take no matter where we are. One might wonder if we ever need to take different actions from a given state. The answer is no, since like as in a search problem, the state contains all the information that we need to act optimally for the future. In more formal speak, the transitions and rewards satisfy the *Markov property*. Every time we end up in a state, we are faced with the exact same problem and therefore should take the same optimal action.

Solutions to:

- **Search problem:** path (sequence of actions)
- **MDPs:** policies (state to action mapping)

Definition: policy. A *policy* π is a mapping from each state $s \in \text{States}$ to an action $a \in \text{Actions}(s)$.

Example: volcano crossing.

s	$\pi(s)$
$(1,1)$	S
$(2,1)$	E
$(3,1)$	N
$\dots$	$\dots$

3.3 *Policy Evaluation*

Now that we've defined an MDP (the input) and a policy (the output), let's turn to defining the evaluation metric for a policy—there are many of them, which one should we choose? Recall that we'd like to maximize the total rewards (utility), but this is a random quantity, so we can't quite do that. Instead, we will instead maximize the *expected utility*, which we will refer to as *value* (of a policy).

Definition: utility. Following a policy yields a *random path*. The *utility* of a policy is the (discounted) sum of the rewards on the path (this is a random quantity).

Definition: value (expected utility). The *value* of a policy is the *expected utility*.

Evaluating a policy: volcano crossing. To get an intuitive feel for the relationship between a value and utility, consider the volcano example. If you press `Run` multiple times, you will get random paths shown on the right leading to different utilities. Note that there is considerable variation in what happens. The expectation of this utility is the *value*. You can run multiple simulations by increasing `numEpisodes`. If you set `numEpisodes` to 1000, then you'll see the average utility converging to the value.

Path	**Utility**	
[in; stay, 4, end]	4	
[in; stay, 4, in; stay, 4, in; stay, 4, end]	12	**Value:** 12
[in; stay, 4, in; stay, 4, end]	8	
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	16	
...	...	

3.3.1 *Discounting*

There is an additional aspect to utility: *discounting*, which captures the fact that a reward today might be worth more than the same reward tomorrow. If the discount γ is small, then we favor the present more and downweight future rewards more. Note that the discounting parameter is applied exponentially to future rewards, so the distant future is always going to have a fairly small contribution to the utility (unless $\gamma = 1$). The terminology, though standard, is slightly confusing: a larger value of the discount parameter γ actually means that the future is discounted less.

Definition: utility. Path: $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \ldots$ (action, reward, new state). The *utility* with discount γ is:

$$u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \cdots$$

- Discount $\gamma = 1$ (save for the future):
  - [stay, stay, stay, stay]: $4 + 4 + 4 + 4 = 16$
- Discount $\gamma = 0$ (live in the moment):
  - [stay, stay, stay, stay]: $4 + 0 \cdot (4 + \cdots) = 4$
- Discount $\gamma = 0.5$ (balanced life):
  - [stay, stay, stay, stay]: $4 + \frac{1}{2} \cdot 4 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 4 = 7.5$

3.3.2 *Policy evaluation*

Associated with any policy π are two important quantities, the value of the policy $V_\pi(s)$ and the Q-value of a policy $Q_\pi(s, a)$. In terms of the MDP graph, one can think of the value $V_\pi(s)$ as labeling the state nodes, and the Q-value $Q_\pi(s, a)$ as labeling the chance nodes. This label refers to the expected utility if we were to start at that node and continue the dynamics of the game.

We will now write down some equations relating value and Q-value. Our eventual goal is to get to an algorithm for computing these values, but as we will see, writing down the relationships gets us most of the way there, just as writing

Definition: value of a policy. Let $V_\pi(s)$ be the expected utility received by following policy π from state s.

Definition: Q-value of a policy. Let $Q_\pi(s, a)$ be the expected utility of taking action a from state s, and then following policy π.

down the recurrence for FutureCost directly lead to a dynamic programming algorithm for acyclic search problems. First, we get $V_\pi(s)$, the value of a state s, by just following the action edge specified by the policy and taking the Q-value $Q_\pi(s, \pi(s))$. (There's also a base case where $\text{IsEnd}(s)$.) Second, we get $Q_\pi(s, a)$ by considering all possible transitions to successor states s' and taking the expectation over the immediate reward $\text{Reward}(s, a, s')$ plus the discounted future reward $\gamma V_\pi(s')$. While we've defined the recurrence for the expected utility directly, we can derive the recurrence by applying the law of total expectation and invoking the Markov property. To do this, we need to set up some random variables: Let s_0 be the initial state, a_1 be the action that we take, r_1 be the reward we obtain, and s_1 be the state we end up in. Also define $u_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$ to be the utility of following policy π from time step t. Then $V_\pi(s) = \mathbb{E}[u_1 \mid s_0 = s]$, which (assuming s is not an end state) in turn equals:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[u_1 \mid s_0 = s] \\ &= \sum_{s'} \mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] \mathbb{E}[u_1 \mid s_1 = s', s_0 = s, a_1 = \pi(s)] \end{aligned}$$

Note that $\mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] = T(s, \pi(s), s')$. Using the fact that $u_1 = r_1 + \gamma u_2$ and taking expectations, we get that:

$$\mathbb{E}[u \mid s_1 = s', s_0 = s, a_1 = \pi(s)] = \text{Reward}(s, \pi(s), s') + \gamma V_\pi(s')$$

The rest follows from algebra.

Plan. Define recurrences relating value and Q-value:

$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise} \end{cases}$$
$$Q_\pi(s,a) = \sum_{s'} T(s,a,s') \left[\text{Reward}(s,a,s') + \gamma V_\pi(s')\right]$$

Example: Dice game. As an example, let's compute the values of the nodes in the dice game for the policy "stay". Note that the recurrence involves both $V_\pi(\text{in})$ on the left-hand side and the right-hand side. At least in this simple example, we can solve this recurrence easily to get the value. Let $\gamma = 1$. Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$\begin{aligned} V_\pi(\text{end}) &= 0 \\ V_\pi(\text{in}) &= Q_\pi(\text{in}, \text{stay}) \\ &= \frac{1}{3}(4 + V_\pi(\text{end})) + \frac{2}{3}(4 + V_\pi(\text{in})) \end{aligned}$$

In this case, can solve in closed form:

$$\begin{aligned} V_\pi(\text{in}) &= \frac{1}{3}4 + \frac{2}{3}(4 + V_\pi(\text{in})) \\ &= 4 + \frac{2}{3}V_\pi(\text{in}) \\ &\implies \frac{1}{3}V_\pi(\text{in}) = 4 \\ V_\pi(\text{in}) &= 12 \end{aligned}$$

But for a much larger MDP with 100000 states, how do we efficiently compute the value of a policy? One option is the following: observe that the recurrences define a system of linear equations, where the variables are $V_\pi(s)$ for each state s and there is an equation for each state. So we could solve the system of linear equations by computing a matrix inverse. However, inverting a 100000×100000 matrix is expensive in general. There is an even simpler approach called *policy evaluation*. We've already seen examples of iterative algorithms in machine learning: the basic idea is to start with something crude, and refine it over time. Policy

iteration starts with a vector of all zeros for the initial values $V_\pi^{(0)}$. Each iteration, we loop over all the states and apply the two recurrences that we had before. The equations look hairier because of the superscript (t), which simply denotes the value of at iteration t of the algorithm.

Key idea: iterative algorithm. Start with arbitrary policy values and repeatedly apply recurrences to converge to true values.

function PolicyEvaluation(π)
  Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.
  for iteration $t = 1, \ldots, t_{\text{PE}}$ **do**
    for each state s **do**

$$V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

Algorithm 3.1. Policy evaluation.

Some implementation notes: a good strategy for determining how many iterations to run policy evaluation is based on how accurate the result is. Rather than set some fixed number of iterations (e.g, 100), we instead set an error tolerance (e.g., $\epsilon = 0.01$), and iterate until the maximum change between values of any state s from one iteration (t) to the previous $(t-1)$ is at most ϵ. The second note is that while the algorithm is stated as computing $V_\pi^{(t)}$ for each iteration t, we actually only need to keep track of the last two values. This is important for saving memory. How many iterations to run (t_{PE})? Repeat until values don't change much:

$$\max_{s \in \text{States}} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

Don't store $V_\pi^{(t)}$ for each iteration t, need only last two: $V_\pi^{(t)}$ and $V_\pi^{(t-1)}$.

Complexity. Computing the running time of policy evaluation is straightforward: for each of the t_{PE} iterations, we need to enumerate through each of the S states, and for each one of those, loop over the successors S'. Note that we don't have a dependence on the number of actions A because we have a fixed policy

$\pi(s)$ and we only need to look at the action specified by the policy. Advanced: Here, we have to iterate t_{PE} time steps to reach a target level of error ϵ. It turns out that t_{PE} doesn't actually have to be very large for very small errors. Specifically, the error decreases exponentially fast as we increase the number of iterations. In other words, to cut the error in half, we only have to run a constant number of more iterations. Advanced: For acyclic graphs (for example, the MDP for Blackjack), we just need to do one iteration (not t_{PE}) provided that we process the nodes in reverse topological order of the graph. This is the same setup as we had for dynamic programming in search problems, only the equations are different.

MDP complexity. $O(t_{\text{PE}}SS')$

- S states
- A actions per state
- S successors (number of s' with $T(s,a,s') > 0$)

Let us run policy evaluation on the dice game. The value converges very quickly to the correct answer. Let π be the stay policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi^{(t)}(\text{end}) = 0$$
$$V_\pi^{(t)}(\text{in}) = \frac{1}{3}(4 + V_\pi^{(t-1)}(\text{end})) + \frac{2}{3}(4 + V_\pi^{(t-1)}(\text{in}))$$

s	end	in	
$V_\pi^{(t)}$	0.00	0.00	($t = 0$ iterations)
$V_\pi^{(t)}$	0.00	4.00	($t = 1$ iterations)
$V_\pi^{(t)}$	0.00	6.67	($t = 2$ iterations)
$V_\pi^{(t)}$	0.00	8.44	($t = 3$ iterations)
$V_\pi^{(t)}$	0.00	12.00	($t = 100$ iterations)

Converges to $V_\pi(\text{in}) = 12$.

3.3.3 Summary

Let's summarize: we have defined an MDP, which we should think of a graph where the nodes are states and chance nodes. Because of randomness, solving an MDP means generating policies, not just paths. A policy is evaluated based on its value: the expected utility obtained over random paths. Finally, we saw that policy evaluation provides a simple way to compute the value of a policy.

- **MDP:** graph with states, chance nodes, transition probabilities, rewards.
- **Policy:** mapping from state to action (solution to MDP).
- **Value of policy:** expected utility over random paths.
- **Policy evaluation:** iterative algorithm to compute value of policy.

3.4 Value Iteration

If we are given a policy π, we now know how to compute its value $V_\pi(\mathrm{s}_{\mathrm{state}})$. So now, we could just enumerate all the policies, compute the value of each one, and take the best policy, but the number of policies is exponential in the number of states (A^S to be exact), so we need something a bit more clever. We will now introduce value iteration, which is an algorithm for finding the best policy. While evaluating a given policy and finding the best policy might seem very different, it turns out that value iteration will look a lot like policy evaluation.

3.4.1 Optimal value and policy

We will write down a bunch of recurrences which look exactly like policy evaluation, but instead of having V_π and Q_π with respect to a fixed policy π, we will have V^* and Q^*, which are with respect to the optimal policy. The goal: try to get directly at maximum expected utility.

Definition: optimal value. The *optimal value* $V^*(s)$ is the maximum value attained by any policy.

The recurrences for V^* and Q^* are identical to the ones for policy evaluation with one difference: in computing V^*, instead of taking the action from the fixed policy π, we take the best action, the one that results in the largest $Q^*(s, a)$.

- Optimal value if take action a in state s:

$$Q^*(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V^*(s')].$$

- Optimal value from state s:

$$V^*(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q^*(s, a) & \text{otherwise} \end{cases}$$

So far, we have focused on computing the value of the optimal policy, but what is the actual policy? It turns out that this is pretty easy to compute. Suppose you're at a state s. $Q^*(s, a)$ tells you the value of taking action a from state s. So the optimal action is simply to take the action a with the largest value of $Q^*(s, a)$. Given Q^*, read off the optimal policy:

$$\pi^*(s) = \arg\max_{a \in \text{Actions}(s)} Q^*(s, a)$$

3.4.2 *Value iteration*

By now, you should be able to go from recurrences to algorithms easily. Following the recipe, we simply iterate some number of iterations, go through each state s and then replace the equality in the recurrence with the assignment operator. Value iteration is also guaranteed to converge to the optimal value. What about the optimal policy? We get it as a byproduct. The optimal value $V^*(s)$ is computed by taking a max over actions. If we take the argmax, then we get the optimal policy $\pi^*(s)$.

3.4.3 *Convergence*

Let us state more formally the conditions under which any of these algorithms that we talked about will work. A sufficient condition is that either the discount γ must be strictly less than 1 or the MDP graph is acyclic. We can reinterpret the

function VALUEITERATION
  Initialize $V^{*(0)}(s) \leftarrow 0$ for all states s.
  for iteration $t = 1, \ldots, t_{\text{VI}}$ **do**
    for each state s **do**
      $$V^{*(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V^{*(t-1)}(s')]}_{Q^{*(t-1)}(s,a)}$$

Time complexity: $O(t_{\text{VI}} SAS')$

Algorithm 3.2. Value iteration [Bellman, 1957].

discount $\gamma < 1$ condition as introducing a new transition from each state to a special end state with probability $(1-\gamma)$, multiplying all the other transition probabilities by γ, and setting the discount to 1. The interpretation is that with probability $1-\gamma$, the MDP terminates at any state. In this view, we just need that a sampled path be finite with probability 1. We won't prove this theorem, but will instead give a counterexample to show that things can go badly if we have a cyclic graph and $\gamma = 1$. In the graph, whatever we initialize value iteration, value iteration will terminate immediately with the same value. In some sense, this isn't really the fault of value iteration, but it's because all paths are of infinite length. In some sense, if you were to simulate from this MDP, you would never terminate, so we would never find out what your utility was at the end.

3.4.4 *Summary of algorithms*

- **Policy evaluation:** (MDP, π) $\rightarrow V_\pi$
- **Value iteration:** MDP $\rightarrow (V^*, \pi^*)$

3.4.5 *Unifying idea*

There are two key ideas in this chapter. First, the policy π, value V_π, and Q-value Q_π are the three key quantities of MDPs, and they are related via a number of recurrences which can be easily gotten by just thinking about their interpretations. Second, given recurrences that depend on each other for the values you're trying

Value iteration: dice game. Let us demonstrate value iteration on the dice game. Initially, the optimal policy is ''quit'', but as we run value iteration longer, it switches to ''stay''.

s	end	in	
$V^{*(t)}$	0.00	0.00	($t = 0$ iterations)
$\pi^*(s)$	—	—	
$V^{*(t)}$	0.00	10.00	($t = 1$ iterations)
$\pi^*(s)$	—	quit	
$V^{*(t)}$	0.00	10.67	($t = 2$ iterations)
$\pi^*(s)$	—	stay	
$V^{*(t)}$	0.00	11.11	($t = 3$ iterations)
$\pi^*(s)$	—	stay	
$V^{*(t)}$	0.00	12.00	($t = 100$ iterations)
$\pi^*(s)$	—	stay	

Theorem: convergence. Suppose either discount $\gamma < 1$, or MDP graph is acyclic. Then value iteration converges to the correct answer.

Example: non-convergence. discount $\gamma = 1$, zero rewards

to compute, it's easy to turn these recurrences into algorithms that iterate between those recurrences until convergence.

Algorithms:

- Search DP computes FutureCost(s)
- Policy evaluation computes policy value $V_\pi(s)$
- Value iteration computes optimal value $V^*(s)$

Recipe:

- Write down recurrence (e.g., $V_\pi(s) = \cdots V_\pi(s') \cdots$)
- Turn into iterative algorithm (replace mathematical equality with assignment operator)

3.4.6 Summary

- *Markov decision processes* (MDPs) cope with uncertainty.
- Solutions are *policies* rather than paths.
- *Policy evaluation* computes policy value (expected utility).
- *Value iteration* computes optimal value (maximum expected utility) and optimal policy.
- **Main technique:** write recurrences $\rightarrow$ algorithm
- **Next chapter:** reinforcement learning—when we don't know rewards, transition probabilities.

References

1. M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019 (cit. on p. v).