4 Linear Neural Networks for Classification

Now that you have worked through all of the mechanics you are ready to apply the skills you have learned to broader kinds of tasks. Even as we pivot towards classification, most of the plumbing remains the same: loading the data, passing it through the model, generating output, calculating the loss, taking gradients with respect to weights, and updating the model. However, the precise form of the targets, the parametrization of the output layer, and the choice of loss function will adapt to suit the *classification* setting.

4.1 Softmax Regression

In Section 3.1, we introduced linear regression, working through implementations from scratch in Section 3.4 and again using high-level APIs of a deep learning framework in Section 3.5 to do the heavy lifting.

Regression is the hammer we reach for when we want to answer *how much?* or *how many?* questions. If you want to predict the number of dollars (price) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you are probably looking for a regression model. However, even within regression models, there are important distinctions. For instance, the price of a house will never be negative and changes might often be *relative* to its baseline price. As such, it might be more effective to regress on the logarithm of the price. Likewise, the number of days a patient spends in hospital is a *discrete nonnegative* random variable. As such, least mean squares might not be an ideal approach either. This sort of time-to-event modeling comes with a host of other complications that are dealt with in a specialized subfield called *survival modeling*.

The point here is not to overwhelm you but just to let you know that there is a lot more to estimation than simply minimizing squared errors. And more broadly, there is a lot more to supervised learning than regression. In this section, we focus on *classification* problems where we put aside *how much?* questions and instead focus on *which category?* questions.

- Does this email belong in the spam folder or the inbox?
- Is this customer more likely to sign up or not to sign up for a subscription service?

- Does this image depict a donkey, a dog, a cat, or a rooster?
- Which movie is Aston most likely to watch next?
- Which section of the book are you going to read next?

Colloquially, machine learning practitioners overload the word *classification* to describe two subtly different problems: (i) those where we are interested only in hard assignments of examples to categories (classes); and (ii) those where we wish to make soft assignments, i.e., to assess the probability that each category applies. The distinction tends to get blurred, in part, because often, even when we only care about hard assignments, we still use models that make soft assignments.

Even more, there are cases where more than one label might be true. For instance, a news article might simultaneously cover the topics of entertainment, business, and space flight, but not the topics of medicine or sports. Thus, categorizing it into one of the above categories on their own would not be very useful. This problem is commonly known as multi-label classification[86]. See Tsoumakas and Katakis (2007) for an overview and Huang *et al.* (2015) for an effective algorithm when tagging images.

86

4.1.1 Classification

To get our feet wet, let's start with a simple image classification problem. Here, each input consists of a 2×2 grayscale image. We can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4. Further, let's assume that each image belongs to one among the categories "cat", "chicken", and "dog".

Next, we have to choose how to represent the labels. We have two obvious choices. Perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent $\{\text{dog}, \text{cat}, \text{chicken}\}$ respectively. This is a great way of *storing* such information on a computer. If the categories had some natural ordering among them, say if we were trying to predict $\{\text{baby}, \text{toddler}, \text{adolescent}, \text{young adult}, \text{adult}, \text{geriatric}\}$, then it might even make sense to cast this as an ordinal regression[87] problem and keep the labels in this format. See Moon *et al.* (2010) for an overview of different types of ranking loss functions and Beutel *et al.* (2014) for a Bayesian approach that addresses responses with more than one mode.

87

In general, classification problems do not come with natural orderings among the classes. Fortunately, statisticians long ago invented a simple way to represent categorical data: the *one-hot encoding*. A one-hot encoding is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0. In our case, a label y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken", and $(0, 0, 1)$ to "dog":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{4.1.1}$$

Linear Model

In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class. To address classification with linear models, we will need as many affine functions as we have outputs. Strictly speaking, we only need one fewer, since the final category has to be the difference between 1 and the sum of the other categories, but for reasons of symmetry we use a slightly redundant parametrization. Each output corresponds to its own affine function. In our case, since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts). This yields:

$$\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,\\
o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,\\
o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.
\end{aligned} \tag{4.1.2}$$

The corresponding neural network diagram is shown in Fig. 4.1.1. Just as in linear regression, we use a single-layer neural network. And since the calculation of each output, o_1, o_2, and o_3, depends on every input, x_1, x_2, x_3, and x_4, the output layer can also be described as a *fully connected layer*.

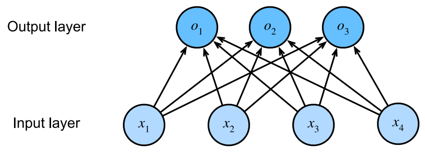


Fig. 4.1.1 Softmax regression is a single-layer neural network.

For a more concise notation we use vectors and matrices: $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$ is much better suited for mathematics and code. Note that we have gathered all of our weights into a 3×4 matrix and all biases $\mathbf{b} \in \mathbb{R}^3$ in a vector.

The Softmax

Assuming a suitable loss function, we could try, directly, to minimize the difference between $\mathbf{o}$ and the labels $\mathbf{y}$. While it turns out that treating classification as a vector-valued regression problem works surprisingly well, it is nonetheless unsatisfactory in the following ways:

- There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.
- There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.

Both aspects render the estimation problem difficult to solve and the solution very brittle to outliers. For instance, if we assume that there is a positive linear dependency between the number of bedrooms and the likelihood that someone will buy a house, the probability might exceed 1 when it comes to buying a mansion! As such, we need a mechanism to "squish" the outputs.

There are many ways we might accomplish this goal. For instance, we could assume that the outputs $\mathbf{o}$ are corrupted versions of $\mathbf{y}$, where the corruption occurs by means of adding noise $\boldsymbol{\epsilon}$ drawn from a normal distribution. In other words, $\mathbf{y} = \mathbf{o} + \boldsymbol{\epsilon}$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. This is the so-called probit model[88], first introduced by Fechner (1860). While appealing, it does not work quite as well nor lead to a particularly nice optimization problem, when compared to the softmax.

88

Another way to accomplish this goal (and to ensure nonnegativity) is to use an exponential function $P(y = i) \propto \exp o_i$. This does indeed satisfy the requirement that the conditional class probability increases with increasing o_i, it is monotonic, and all probabilities are nonnegative. We can then transform these values so that they add up to 1 by dividing each by their sum. This process is called *normalization*. Putting these two pieces together gives us the *softmax* function:

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}. \tag{4.1.3}$$

Note that the largest coordinate of $\mathbf{o}$ corresponds to the most likely class according to $\hat{\mathbf{y}}$. Moreover, because the softmax operation preserves the ordering among its arguments, we do not need to compute the softmax to determine which class has been assigned the highest probability. Thus,

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{4.1.4}$$

The idea of a softmax dates back to Gibbs (1902), who adapted ideas from physics. Dating even further back, Boltzmann, the father of modern statistical physics, used this trick to model a distribution over energy states in gas molecules. In particular, he discovered that the prevalence of a state of energy in a thermodynamic ensemble, such as the molecules in a gas, is proportional to $\exp(-E/kT)$. Here, E is the energy of a state, T is the temperature, and k is the Boltzmann constant. When statisticians talk about increasing or decreasing the "temperature" of a statistical system, they refer to changing T in order to favor lower or higher energy states. Following Gibbs' idea, energy equates to error. Energy-based models (Ranzato *et al.*, 2007) use this point of view when describing problems in deep learning.

Vectorization

To improve computational efficiency, we vectorize calculations in minibatches of data. Assume that we are given a minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ of n examples with dimensionality (number of inputs) d. Moreover, assume that we have q categories in the output. Then the weights satisfy $\mathbf{W} \in \mathbb{R}^{d \times q}$ and the bias satisfies $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned} \tag{4.1.5}$$

This accelerates the dominant operation into a matrix–matrix product $\mathbf{XW}$. Moreover, since each row in $\mathbf{X}$ represents a data example, the softmax operation itself can be computed *rowwise*: for each row of $\mathbf{O}$, exponentiate all entries and then normalize them by the sum. Note, though, that care must be taken to avoid exponentiating and taking logarithms of large numbers, since this can cause numerical overflow or underflow. Deep learning frameworks take care of this automatically.

4.1.2 Loss Function

Now that we have a mapping from features $\mathbf{x}$ to probabilities $\hat{\mathbf{y}}$, we need a way to optimize the accuracy of this mapping. We will rely on maximum likelihood estimation, the very same method that we encountered when providing a probabilistic justification for the mean squared error loss in Section 3.1.3.

Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as the (estimated) conditional probabilities of each class, given any input $\mathbf{x}$, such as $\hat{y}_1 = P(y = \text{cat} \mid \mathbf{x})$. In the following we assume that for a dataset with features $\mathbf{X}$ the labels $\mathbf{Y}$ are represented using a one-hot encoding label vector. We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{4.1.6}$$

We are allowed to use the factorization since we assume that each label is drawn independently from its respective distribution $P(\mathbf{y} \mid \mathbf{x}^{(i)})$. Since maximizing the product of terms is awkward, we take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{4.1.7}$$

where for any pair of label $\mathbf{y}$ and model prediction $\hat{\mathbf{y}}$ over q classes, the loss function l is

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{4.1.8}$$

For reasons explained later on, the loss function in (4.1.8) is commonly called the *cross-entropy loss*. Since $\mathbf{y}$ is a one-hot vector of length q, the sum over all its coordinates j vanishes for all but one term. Note that the loss $l(\mathbf{y}, \hat{\mathbf{y}})$ is bounded from below by 0 whenever $\hat{\mathbf{y}}$ is a probability vector: no single entry is larger than 1, hence their negative logarithm cannot be lower than 0; $l(\mathbf{y}, \hat{\mathbf{y}}) = 0$ only if we predict the actual label with *certainty*. This can never happen for any finite setting of the weights because taking a softmax output towards 1 requires taking the corresponding input o_i to infinity (or all other outputs o_j for $j \neq i$ to negative infinity). Even if our model could assign an output probability of 0, any error made when assigning such high confidence would incur infinite loss ($-\log 0 = \infty$).

Softmax and Cross-Entropy Loss

Since the softmax function and the corresponding cross-entropy loss are so common, it is worth understanding a bit better how they are computed. Plugging (4.1.3) into the definition of the loss in (4.1.8) and using the definition of the softmax we obtain

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= - \sum_{j=1}^q y_j \log \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} \\ &= \sum_{j=1}^q y_j \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j \\ &= \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j. \end{aligned} \tag{4.1.9}$$

To understand a bit better what is going on, consider the derivative with respect to any logit o_j. We get

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} - y_j = \mathrm{softmax}(\mathbf{o})_j - y_j. \tag{4.1.10}$$

In other words, the derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and what actually happened, as expressed by elements in the one-hot label vector. In this sense, it is very similar to what we saw in regression, where the gradient was the difference between the observation y and estimate $\hat{y}$. This is not a coincidence. In any exponential family model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Now consider the case where we observe not just a single outcome but an entire distribution over outcomes. We can use the same representation as before for the label $\mathbf{y}$. The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l in (4.1.8) still works well, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels. This loss is called the *cross-entropy loss* and it is one of the most commonly used losses for classification problems. We can demystify the name by introducing just the basics of information theory. In a nutshell, it measures the number of bits needed to encode what we see, $\mathbf{y}$, relative to what we predict that should happen, $\hat{\mathbf{y}}$. We provide a very basic explanation in the following. For further details on information theory see Cover and Thomas (1999) or MacKay (2003).

4.1.3 Information Theory Basics

Many deep learning papers use intuition and terms from information theory. To make sense of them, we need some common language. This is a survival guide. *Information theory* deals with the problem of encoding, decoding, transmitting, and manipulating information (also known as data).

Entropy

The central idea in information theory is to quantify the amount of information contained in data. This places a limit on our ability to compress data. For a distribution P its *entropy*, $H[P]$, is defined as:

$$H[P] = \sum_j -P(j) \log P(j). \tag{4.1.11}$$

One of the fundamental theorems of information theory states that in order to encode data drawn randomly from the distribution P, we need at least $H[P]$ "nats" to encode it (Shannon, 1948). If you wonder what a "nat" is, it is the equivalent of bit but when using a code with base e rather than one with base 2. Thus, one nat is $\frac{1}{\log(2)} \approx 1.44$ bit.

Surprisal

You might be wondering what compression has to do with prediction. Imagine that we have a stream of data that we want to compress. If it is always easy for us to predict the next token, then this data is easy to compress. Take the extreme example where every token in the stream always takes the same value. That is a very boring data stream! And not only it is boring, but it is also easy to predict. Because the tokens are always the same, we do not have to transmit any information to communicate the contents of the stream. Easy to predict, easy to compress.

However if we cannot perfectly predict every event, then we might sometimes be surprised. Our surprise is greater when an event is assigned lower probability. Claude Shannon settled on $\log \frac{1}{P(j)} = -\log P(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $P(j)$. The entropy defined in (4.1.11) is then the *expected surprisal* when one assigned the correct probabilities that truly match the data-generating process.

Cross-Entropy Revisited

So if entropy is the level of surprise experienced by someone who knows the true probability, then you might be wondering, what is cross-entropy? The cross-entropy *from P to Q*, denoted $H(P, Q)$, is the expected surprisal of an observer with subjective probabilities Q upon seeing data that was actually generated according to probabilities P. This is given by $H(P, Q) \overset{\text{def}}{=} \sum_j -P(j) \log Q(j)$. The lowest possible cross-entropy is achieved when $P = Q$. In this case, the cross-entropy from P to Q is $H(P, P) = H(P)$.

In short, we can think of the cross-entropy classification objective in two ways: (i) as maximizing the likelihood of the observed data; and (ii) as minimizing our surprisal (and thus the number of bits) required to communicate the labels.

4.1.4 Summary and Discussion

In this section, we encountered the first nontrivial loss function, allowing us to optimize over *discrete* output spaces. Key in its design was that we took a probabilistic approach, treating discrete categories as instances of draws from a probability distribution. As a side effect, we encountered the softmax, a convenient activation function that transforms outputs of an ordinary neural network layer into valid discrete probability distributions. We saw that the derivative of the cross-entropy loss when combined with softmax behaves very similarly to the derivative of squared error; namely by taking the difference between the expected behavior and its prediction. And, while we were only able to scratch the very surface of it, we encountered exciting connections to statistical physics and information theory.

While this is enough to get you on your way, and hopefully enough to whet your appetite, we hardly dived deep here. Among other things, we skipped over computational considerations. Specifically, for any fully connected layer with d inputs and q outputs, the parametrization and computational cost is $O(dq)$, which can be prohibitively high in practice. Fortunately, this cost of transforming d inputs into q outputs can be reduced through approximation and compression. For instance Deep Fried Convnets (Yang *et al.*, 2015) uses a combination of permutations, Fourier transforms, and scaling to reduce the cost from quadratic to log-linear. Similar techniques work for more advanced structural matrix approximations (Sindhwani *et al.*, 2015). Lastly, we can use quaternion-like decompositions to reduce the cost to $O(\frac{dq}{n})$, again if we are willing to trade off a small amount of accuracy for computational and storage cost (Zhang *et al.*, 2021) based on a compression factor n. This is an active area of research. What makes it challenging is that we do not necessarily strive for the most compact representation or the smallest number of floating point operations but rather for the solution that can be executed most efficiently on modern GPUs.

4.1.5 Exercises

1. We can explore the connection between exponential families and softmax in some more depth.
    1. Compute the second derivative of the cross-entropy loss $l(\mathbf{y}, \hat{\mathbf{y}})$ for softmax.
    2. Compute the variance of the distribution given by $\mathrm{softmax}(\mathbf{o})$ and show that it matches the second derivative computed above.
2. Assume that we have three classes which occur with equal probability, i.e., the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
    1. What is the problem if we try to design a binary code for it?
    2. Can you design a better code? Hint: what happens if we try to encode two independent observations? What if we encode n observations jointly?
3. When encoding signals transmitted over a physical wire, engineers do not always use binary codes. For instance, PAM-3[89] uses three signal levels $\{-1, 0, 1\}$ as opposed to two levels $\{0, 1\}$. How many ternary units do you need to transmit an integer in the range $\{0, \ldots, 7\}$? Why might this be a better idea in terms of electronics?
4. The Bradley–Terry model[90] uses a logistic model to capture preferences. For a user to

89

90

choose between apples and oranges one assumes scores o_{apple} and o_{orange}. Our requirements are that larger scores should lead to a higher likelihood in choosing the associated item and that the item with the largest score is the most likely one to be chosen (Bradley and Terry, 1952).

   1. Prove that softmax satisfies this requirement.
   2. What happens if you want to allow for a default option of choosing neither apples nor oranges? Hint: now the user has three choices.

5. Softmax gets its name from the following mapping: $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
   1. Prove that $\text{RealSoftMax}(a, b) > \max(a, b)$.
   2. How small can you make the difference between both functions? Hint: without loss of generality you can set $b = 0$ and $a \geq b$.
   3. Prove that this holds for $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.
   4. Show that for $\lambda \rightarrow \infty$ we have $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \rightarrow \max(a, b)$.
   5. Construct an analogous softmin function.
   6. Extend this to more than two numbers.
6. The function $g(\mathbf{x}) \overset{\text{def}}{=} \log \sum_i \exp x_i$ is sometimes also referred to as the log-partition function[91].
   1. Prove that the function is convex. Hint: to do so, use the fact that the first derivative amounts to the probabilities from the softmax function and show that the second derivative is the variance.
   2. Show that g is translation invariant, i.e., $g(\mathbf{x} + b) = g(\mathbf{x})$.
   3. What happens if some of the coordinates x_i are very large? What happens if they're all very small?
   4. Show that if we choose $b = \max_i x_i$ we end up with a numerically stable implementation.
7. Assume that we have some probability distribution P. Suppose we pick another distribution Q with $Q(i) \propto P(i)^\alpha$ for $\alpha > 0$.
   1. Which choice of α corresponds to doubling the temperature? Which choice corresponds to halving it?
   2. What happens if we let the temperature approach 0?
   3. What happens if we let the temperature approach ∞?

Discussions[92].

91

92

4.2 The Image Classification Dataset

93

One widely used dataset for image classification is the MNIST dataset[93] (LeCun *et al.*, 1998) of handwritten digits. At the time of its release in the 1990s it posed a formidable challenge to most machine learning algorithms, consisting of 60,000 images of 28×28 pixels resolution (plus a test dataset of 10,000 images). To put things into perspective, back in 1995, a Sun SPARCStation 5 with a whopping 64MB of RAM and a blistering 5 MFLOPs was considered state of the art equipment for machine learning at AT&T Bell Laboratories. Achieving high accuracy on digit recognition was a key component in automating letter sorting for the USPS in the 1990s. Deep networks such as LeNet-5 (LeCun *et al.*, 1995), support vector machines with invariances (Schölkopf *et al.*, 1996), and tangent distance classifiers (Simard *et al.*, 1998) all could reach error rates below 1%.

For over a decade, MNIST served as *the* point of reference for comparing machine learning algorithms. While it had a good run as a benchmark dataset, even simple models by today's standards achieve classification accuracy over 95%, making it unsuitable for distinguishing between strong models and weaker ones. Even more, the dataset allows for *very* high levels of accuracy, not typically seen in many classification problems. This skewed algorithmic development towards specific families of algorithms that can take advantage of clean datasets, such as active set methods and boundary-seeking active set algorithms. Today, MNIST serves as more of a sanity check than as a benchmark. ImageNet (Deng *et al.*, 2009) poses a much more relevant challenge. Unfortunately, ImageNet is too large for many of the examples and illustrations in this book, as it would take too long to train to make the examples interactive. As a substitute we will focus our discussion in the coming sections on the qualitatively similar, but much smaller Fashion-MNIST dataset (Xiao *et al.*, 2017) which was released in 2017. It contains images of 10 categories of clothing at 28×28 pixels resolution.

```python
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

4.2.1 Loading the Dataset

Since the Fashion-MNIST dataset is so useful, all major frameworks provide preprocessed versions of it. We can download and read it into memory using built-in framework utilities.

```
class FashionMNIST(d2l.DataModule):  #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
```

Fashion-MNIST consists of images from 10 categories, each represented by 6000 images in the training dataset and by 1000 in the test dataset. A *test dataset* is used for evaluating model performance (it must not be used for training). Consequently the training set and the test set contain 60,000 and 10,000 images, respectively.

```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

```
(60000, 10000)
```

The images are grayscale and upscaled to 32×32 pixels in resolution above. This is similar to the original MNIST dataset which consisted of (binary) black and white images. Note, though, that most modern image data has three channels (red, green, blue) and that hyperspectral images can have in excess of 100 channels (the HyMap sensor has 126 channels). By convention we store an image as a $c \times h \times w$ tensor, where c is the number of color channels, h is the height and w is the width.

```
data.train[0][0].shape
```

```
torch.Size([1, 32, 32])
```

The categories of Fashion-MNIST have human-understandable names. The following convenience method converts between numeric labels and their names.

```
@d2l.add_to_class(FashionMNIST)  #@save
def text_labels(self, indices):
    """Return text labels."""
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

4.2.2 Reading a Minibatch

To make our life easier when reading from the training and test sets, we use the built-in data iterator rather than creating one from scratch. Recall that at each iteration, a data iterator reads a minibatch of data with size `batch_size`. We also randomly shuffle the examples for the training data iterator.

```
@d2l.add_to_class(FashionMNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

To see how this works, let's load a minibatch of images by invoking the `train_dataloader` method. It contains 64 images.

```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

Let's look at the time it takes to read the images. Even though it is a built-in loader, it is not blazingly fast. Nonetheless, this is sufficient since processing images with a deep network takes quite a bit longer. Hence it is good enough that training a network will not be I/O constrained.

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'4.69 sec'
```

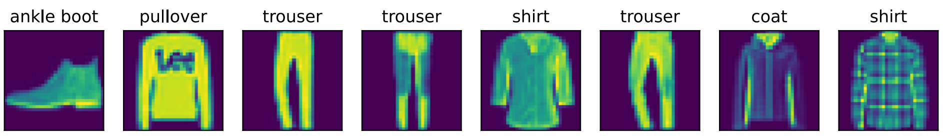
4.2.3 Visualization

We will often be using the Fashion-MNIST dataset. A convenience function `show_images` can be used to visualize the images and the associated labels. Skipping implementation details, we just show the interface below: we only need to know how to invoke `d2l.show_images` rather than how it works for such utility functions.

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    raise NotImplementedError
```

Let's put it to good use. In general, it is a good idea to visualize and inspect data that you are training on. Humans are very good at spotting oddities and because of that, visualization serves as an additional safeguard against mistakes and errors in the design of experiments. Here are the images and their corresponding labels (in text) for the first few examples in the training dataset.

```
@d2l.add_to_class(FashionMNIST)  #@save
def visualize(self, batch, nrows=1, ncols=8, labels=[]):
    X, y = batch
    if not labels:
        labels = self.text_labels(y)
    d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```



We are now ready to work with the Fashion-MNIST dataset in the sections that follow.

4.2.4 Summary

We now have a slightly more realistic dataset to use for classification. Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories. We will use this dataset in subsequent sections and chapters to evaluate various network designs, from a simple linear model to advanced residual networks. As we commonly do with images, we read them as a tensor of shape (batch size, number of channels, height, width). For now, we only have one channel as the images are grayscale (the visualization above uses a false color palette for improved visibility).

Lastly, data iterators are a key component for efficient performance. For instance, we might use GPUs for efficient image decompression, video transcoding, or other preprocessing. Whenever possible, you should rely on well-implemented data iterators that exploit high-performance computing to avoid slowing down your training loop.

4.2.5 Exercises

1. Does reducing the `batch_size` (for instance, to 1) affect the reading performance?
2. The data iterator performance is important. Do you think the current implementation is fast enough? Explore various options to improve it. Use a system profiler to find out where the bottlenecks are.
3. Check out the framework's online API documentation. Which other datasets are available?

94

Discussions[94].

4.3 The Base Classification Model

You may have noticed that the implementations from scratch and the concise implementation using framework functionality were quite similar in the case of regression. The same is true for classification. Since many models in this book deal with classification, it is worth adding functionalities to support this setting specifically. This section provides a base class for classification models to simplify future code.

```python
import torch
from d2l import torch as d2l
```

4.3.1 The `Classifier` Class

We define the `Classifier` class below. In the `validation_step` we report both the loss value and the classification accuracy on a validation batch. We draw an update for every `num_val_batches` batches. This has the benefit of generating the averaged loss and accuracy on the whole validation data. These average numbers are not exactly correct if the final batch contains fewer examples, but we ignore this minor difference to keep the code simple.

```python
class Classifier(d2l.Module):  #@save
    """The base class of classification models."""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
```

By default we use a stochastic gradient descent optimizer, operating on minibatches, just as we did in the context of linear regression.

```python
@d2l.add_to_class(d2l.Module)  #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

4.3.2 Accuracy

Given the predicted probability distribution `y_hat`, we typically choose the class with the highest predicted probability whenever we must output a hard prediction. Indeed, many applications require that we make a choice. For instance, Gmail must categorize an email into "Primary", "Social", "Updates", "Forums", or "Spam". It might estimate probabilities internally, but at the end of the day it has to choose one among the classes.

When predictions are consistent with the label class `y`, they are correct. The classification accuracy is the fraction of all predictions that are correct. Although it can be difficult to optimize accuracy directly (it is not differentiable), it is often the performance measure that we care about the most. It is often *the* relevant quantity in benchmarks. As such, we will nearly always report it when training classifiers.

Accuracy is computed as follows. First, if `y_hat` is a matrix, we assume that the second dimension stores prediction scores for each class. We use `argmax` to obtain the predicted class by the index for the largest entry in each row. Then we compare the predicted class with the ground truth `y` elementwise. Since the equality operator == is sensitive to data types, we convert `y_hat`'s data type to match that of `y`. The result is a tensor containing entries of 0 (false) and 1 (true). Taking the sum yields the number of correct predictions.

```python
@d2l.add_to_class(Classifier)  #@save
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the number of correct predictions."""
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    preds = Y_hat.argmax(axis=1).type(Y.dtype)
    compare = (preds == Y.reshape(-1)).type(torch.float32)
    return compare.mean() if averaged else compare
```

4.3.3 Summary

Classification is a sufficiently common problem that it warrants its own convenience functions. Of central importance in classification is the *accuracy* of the classifier. Note that while we often care primarily about accuracy, we train classifiers to optimize a variety of other objectives for statistical and computational reasons. However, regardless of which loss function was minimized during training, it is useful to have a convenience method for assessing the accuracy of our classifier empirically.

4.3.4 Exercises

1. Denote by L_v the validation loss, and let L_v^q be its quick and dirty estimate computed by the loss function averaging in this section. Lastly, denote by l_v^b the loss on the last minibatch. Express L_v in terms of L_v^q, l_v^b, and the sample and minibatch sizes.
2. Show that the quick and dirty estimate L_v^q is unbiased. That is, show that $E[L_\mathrm{v}] = E[L_\mathrm{v}^\mathrm{q}]$. Why would you still want to use L_v instead?
3. Given a multiclass classification loss, denoting by $l(y, y')$ the penalty of estimating y' when we see y and given a probabilty $p(y \mid x)$, formulate the rule for an optimal selection of y'. Hint: express the expected loss, using l and $p(y \mid x)$.

95

Discussions[95].

4.4 Softmax Regression Implementation from Scratch

Because softmax regression is so fundamental, we believe that you ought to know how to implement it yourself. Here, we limit ourselves to defining the softmax-specific aspects of the model and reuse the other components from our linear regression section, including the training loop.

```
import torch
from d2l import torch as d2l
```

4.4.1 The Softmax

Let's begin with the most important part: the mapping from scalars to probabilities. For a refresher, recall the operation of the sum operator along specific dimensions in a tensor, as discussed in Section 2.3.6 and Section 2.3.7. Given a matrix `X` we can sum over all elements (by default) or only over elements in the same axis. The `axis` variable lets us compute row and column sums:

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

Computing the softmax requires three steps: (i) exponentiation of each term; (ii) a sum over each row to compute the normalization constant for each example; (iii) division of each row by its normalization constant, ensuring that the result sums to 1:

$$\mathrm{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{4.4.1}$$

The (logarithm of the) denominator is called the (log) *partition function*. It was introduced in statistical physics[96] to sum over all possible states in a thermodynamic ensemble. The implementation is straightforward:

96

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition  # The broadcasting mechanism is applied here
```

For any input `X`, we turn each element into a nonnegative number. Each row sums up to 1, as is required for a probability. Caution: the code above is *not* robust against very large or very small arguments. While it is sufficient to illustrate what is happening, you should *not* use this code verbatim for any serious purpose. Deep learning frameworks have such protections built in and we will be using the built-in softmax going forward.

```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2511, 0.1417, 0.1158, 0.2529, 0.2385],
         [0.2004, 0.1419, 0.1957, 0.2504, 0.2117]]),
 tensor([1., 1.]))
```

4.4.2 The Model

We now have everything that we need to implement the softmax regression model. As in our linear regression example, each instance will be represented by a fixed-length vector. Since the raw data here consists of 28×28 pixel images, we flatten each image, treating them as vectors of length 784. In later chapters, we will introduce convolutional neural networks, which exploit the spatial structure in a more satisfying way.

In softmax regression, the number of outputs from our network should be equal to the number of classes. Since our dataset has 10 classes, our network has an output dimension of 10. Consequently, our weights constitute a 784×10 matrix plus a 1×10 row vector for the biases. As with linear regression, we initialize the weights `W` with Gaussian noise. The biases are initialized as zeros.

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs),
                              requires_grad=True)
        self.b = torch.zeros(num_outputs, requires_grad=True)

    def parameters(self):
        return [self.W, self.b]
```

The code below defines how the network maps each input to an output. Note that we flatten each 28×28 pixel image in the batch into a vector using `reshape` before passing the data through our model.

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

4.4.3 The Cross-Entropy Loss

Next we need to implement the cross-entropy loss function (introduced in Section 4.1.2). This may be the most common loss function in all of deep learning. At the moment, applications of deep learning easily cast as classification problems far outnumber those better treated as regression problems.

Recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label. For efficiency we avoid Python for-loops and use indexing instead. In particular, the one-hot encoding in $\mathbf{y}$ allows us to select the matching terms in $\hat{\mathbf{y}}$.

To see this in action we create sample data `y_hat` with 2 examples of predicted probabilities over 3 classes and their corresponding labels `y`. The correct labels are 0 and 2 respectively (i.e., the first and third class). Using `y` as the indices of the probabilities in `y_hat`, we can pick out terms efficiently.

```python
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

Now we can implement the cross-entropy loss function by averaging over the logarithms of the selected probabilities.

```python
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

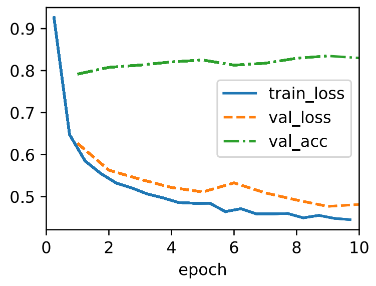
```
tensor(1.4979)
```

```python
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

4.4.4 Training

We reuse the `fit` method defined in Section 3.4 to train the model with 10 epochs. Note that the number of epochs (`max_epochs`), the minibatch size (`batch_size`), and learning rate (`lr`) are adjustable hyperparameters. That means that while these values are not learned during our primary training loop, they still influence the performance of our model, both vis-à-vis training and generalization performance. In practice you will want to choose these values based on the *validation* split of the data and then, ultimately, to evaluate your final model on the *test* split. As discussed in Section 3.6.3, we will regard the test data of Fashion-MNIST as the validation set, thus reporting validation loss and validation accuracy on this split.

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



4.4.5 Prediction

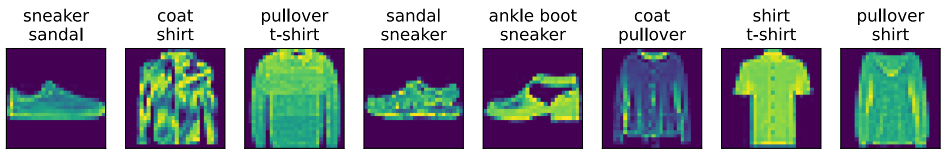
Now that training is complete, our model is ready to classify some images.

```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

```
torch.Size([256])
```

We are more interested in the images we label *incorrectly*. We visualize them by comparing their actual labels (first line of text output) with the predictions from the model (second line of text output).

```
wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```



4.4.6 Summary

By now we are starting to get some experience with solving linear regression and classification problems. With it, we have reached what would arguably be the state of the art of

1960–1970s of statistical modeling. In the next section, we will show you how to leverage deep learning frameworks to implement this model much more efficiently.

4.4.7 Exercises

1. In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. As discussed in Section 4.1 this can cause numerical instabilities.
    1. Test whether `softmax` still works correctly if an input has a value of 100.
    2. Test whether `softmax` still works correctly if the largest of all inputs is smaller than -100?
    3. Implement a fix by looking at the value relative to the largest entry in the argument.
2. Implement a `cross_entropy` function that follows the definition of the cross-entropy loss function $\sum_i y_i \log \hat{y}_i$.
    1. Try it out in the code example of this section.
    2. Why do you think it runs more slowly?
    3. Should you use it? When would it make sense to?
    4. What do you need to be careful of? Hint: consider the domain of the logarithm.
3. Is it always a good idea to return the most likely label? For example, would you do this for medical diagnosis? How would you try to address this?
4. Assume that we want to use softmax regression to predict the next word based on some features. What are some problems that might arise from a large vocabulary?
5. Experiment with the hyperparameters of the code in this section. In particular:
    1. Plot how the validation loss changes as you change the learning rate.
    2. Do the validation and training loss change as you change the minibatch size? How large or small do you need to go before you see an effect?

97

Discussions[97].

4.5 Concise Implementation of Softmax Regression

Just as high-level deep learning frameworks made it easier to implement linear regression (see Section 3.5), they are similarly convenient here.

```python
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

4.5.1 Defining the Model

As in Section 3.5, we construct our fully connected layer using the built-in layer. The built-in `__call__` method then invokes `forward` whenever we need to apply the network to some input.

We use a `Flatten` layer to convert the fourth-order tensor `X` to second order by keeping the dimensionality along the first axis unchanged.

```python
class SoftmaxRegression(d2l.Classifier):  #@save
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(),
                                 nn.LazyLinear(num_outputs))

    def forward(self, X):
        return self.net(X)
```

4.5.2 Softmax Revisited

In Section 4.4 we calculated our model's output and applied the cross-entropy loss. While this is perfectly reasonable mathematically, it is risky computationally, because of numerical underflow and overflow in the exponentiation.

Recall that the softmax function computes probabilities via $\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$. If some of the o_k are very large, i.e., very positive, then $\exp(o_k)$ might be larger than the largest number we can have for certain data types. This is called *overflow*. Likewise, if every argument is a very large negative number, we will get *underflow*. For instance, single precision floating point numbers approximately cover the range of 10^{-38} to 10^{38}. As such, if the largest term in $\mathbf{o}$ lies outside the interval $[-90, 90]$, the result will not be stable. A way round this problem is to subtract $\bar{o} \stackrel{\text{def}}{=} \max_k o_k$ from all entries:

$$\hat{y}_j = \frac{\exp o_j}{\sum_k \exp o_k} = \frac{\exp(o_j - \bar{o}) \exp \bar{o}}{\sum_k \exp(o_k - \bar{o}) \exp \bar{o}} = \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})}. \tag{4.5.1}$$

By construction we know that $o_j - \bar{o} \leq 0$ for all j. As such, for a q-class classification problem, the denominator is contained in the interval $[1, q]$. Moreover, the numerator never exceeds 1, thus preventing numerical overflow. Numerical underflow only occurs when $\exp(o_j - \bar{o})$ numerically evaluates as 0. Nonetheless, a few steps down the road we might find ourselves in trouble when we want to compute $\log \hat{y}_j$ as $\log 0$. In particular, in backpropagation, we might find ourselves faced with a screenful of the dreaded `NaN` (Not a Number) results.

Fortunately, we are saved by the fact that even though we are computing exponential functions, we ultimately intend to take their log (when calculating the cross-entropy loss). By combining softmax and cross-entropy, we can escape the numerical stability issues altogether. We have:

$$\log \hat{y}_j = \log \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})} = o_j - \bar{o} - \log \sum_k \exp(o_k - \bar{o}). \tag{4.5.2}$$

This avoids both overflow and underflow. We will want to keep the conventional softmax function handy in case we ever want to evaluate the output probabilities by our model. But instead of passing softmax probabilities into our new loss function, we just pass the logits and compute the softmax and its log all at once inside the cross-entropy loss function, which does smart things like the "LogSumExp trick"[98].

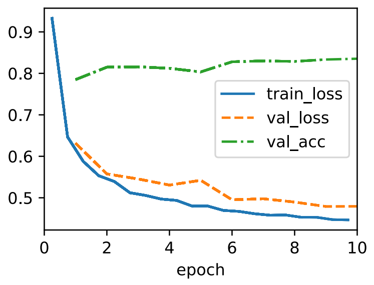
98

```
@d2l.add_to_class(d2l.Classifier)  #@save
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    Y = Y.reshape((-1,))
    return F.cross_entropy(
        Y_hat, Y, reduction='mean' if averaged else 'none')
```

4.5.3 Training

Next we train our model. We use Fashion-MNIST images, flattened to 784-dimensional feature vectors.

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



As before, this algorithm converges to a solution that is reasonably accurate, albeit this time with fewer lines of code than before.

4.5.4 Summary

High-level APIs are very convenient at hiding from their user potentially dangerous aspects, such as numerical stability. Moreover, they allow users to design models concisely with very few lines of code. This is both a blessing and a curse. The obvious benefit is that it makes things highly accessible, even to engineers who never took a single class of statistics in their life (in fact, they are part of the target audience of the book). But hiding the sharp edges also comes with a price: a disincentive to add new and different components on your own, since there is little muscle memory for doing it. Moreover, it makes it more difficult to *fix* things whenever the protective padding of a framework fails to cover all the corner cases entirely. Again, this is due to lack of familiarity.

As such, we strongly urge you to review *both* the bare bones and the elegant versions of many of the implementations that follow. While we emphasize ease of understanding, the implementations are nonetheless usually quite performant (convolutions are the big exception here). It is our intention to allow you to build on these when you invent something new that no framework can give you.

4.5.5 Exercises

1. Deep learning uses many different number formats, including FP64 double precision (used extremely rarely), FP32 single precision, BFLOAT16 (good for compressed representations), FP16 (very unstable), TF32 (a new format from NVIDIA), and INT8. Compute the smallest and largest argument of the exponential function for which the result does not lead to numerical underflow or overflow.
2. INT8 is a very limited format consisting of nonzero numbers from 1 to 255. How could you extend its dynamic range without using more bits? Do standard multiplication and addition still work?
3. Increase the number of epochs for training. Why might the validation accuracy decrease after a while? How could we fix this?
4. What happens as you increase the learning rate? Compare the loss curves for several learning rates. Which one works better? When?

Discussions[99].

99

4.6 Generalization in Classification

So far, we have focused on how to tackle multiclass classification problems by training (linear) neural networks with multiple outputs and softmax functions. Interpreting our model's outputs as probabilistic predictions, we motivated and derived the cross-entropy loss function, which calculates the negative log likelihood that our model (for a fixed set of parameters) assigns to the actual labels. And finally, we put these tools into practice by fitting our model to the training set. However, as always, our goal is to learn *general patterns*, as assessed empirically on previously unseen data (the test set). High accuracy on the training set means nothing. Whenever each of our inputs is unique (and indeed this is true for most high-dimensional datasets), we can attain perfect accuracy on the training set by just memorizing the dataset on the first training epoch, and subsequently looking up the label whenever we see a new image. And yet, memorizing the exact labels associated with the exact training examples does not tell us how to classify new examples. Absent further guidance, we might have to fall back on random guessing whenever we encounter new examples.

A number of burning questions demand immediate attention:

1. How many test examples do we need to give a good estimate of the accuracy of our classifiers on the underlying population?
2. What happens if we keep evaluating models on the same test repeatedly?
3. Why should we expect that fitting our linear models to the training set should fare any better than our naive memorization scheme?

Whereas Section 3.6 introduced the basics of overfitting and generalization in the context of linear regression, this chapter will go a little deeper, introducing some of the foundational ideas of statistical learning theory. It turns out that we often can guarantee generalization *a priori*: for many models, and for any desired upper bound on the generalization gap ϵ, we can often determine some required number of samples n such that if our training set contains at least n samples, our empirical error will lie within ϵ of the true error, *for any data generating distribution*. Unfortunately, it also turns out that while these sorts of guarantees provide a profound set of intellectual building blocks, they are of limited practical utility to the deep learning practitioner. In short, these guarantees suggest that ensuring generalization of deep neural networks *a priori* requires an absurd number of examples (perhaps trillions or more), even when we find that, on the tasks we care about, deep neural networks typically generalize remarkably well with far fewer examples (thousands). Thus deep learning practitioners often forgo *a priori* guarantees altogether, instead employing methods that have generalized well on similar problems in the past, and certifying generalization *post hoc* through empirical evaluations. When we get to Chapter 5, we will revisit generalization and provide a light introduction to the vast scientific literature that has sprung in attempts to explain why deep neural networks generalize in practice.

4.6.1 The Test Set

Since we have already begun to rely on test sets as the gold standard method for assessing generalization error, let's get started by discussing the properties of such error estimates. Let's focus on a fixed classifier f, without worrying about how it was obtained. Moreover suppose that we possess a *fresh* dataset of examples $\mathcal{D} = (\mathbf{x}^{(i)}, y^{(i)})_{i=1}^n$ that were not used to train the classifier f. The *empirical error* of our classifier f on $\mathcal{D}$ is simply the fraction of instances for which the prediction $f(\mathbf{x}^{(i)})$ disagrees with the true label $y^{(i)}$, and is given by the following expression:

$$\epsilon_\mathcal{D}(f) = \frac{1}{n}\sum_{i=1}^n \mathbf{1}(f(\mathbf{x}^{(i)}) \neq y^{(i)}). \tag{4.6.1}$$

By contrast, the *population error* is the *expected* fraction of examples in the underlying population (some distribution $P(X, Y)$ characterized by probability density function $p(\mathbf{x}, y)$)

for which our classifier disagrees with the true label:

$$\epsilon(f) = E_{(\mathbf{x},y)\sim P}\mathbf{1}(f(\mathbf{x}) \neq y) = \int\int \mathbf{1}(f(\mathbf{x}) \neq y)p(\mathbf{x}, y)\; d\mathbf{x}dy. \tag{4.6.2}$$

While $\epsilon(f)$ is the quantity that we actually care about, we cannot observe it directly, just as we cannot directly observe the average height in a large population without measuring every single person. We can only estimate this quantity based on samples. Because our test set $\mathcal{D}$ is statistically representative of the underlying population, we can view $\epsilon_{\mathcal{D}}(f)$ as a statistical estimator of the population error $\epsilon(f)$. Moreover, because our quantity of interest $\epsilon(f)$ is an expectation (of the random variable $\mathbf{1}(f(X) \neq Y)$) and the corresponding estimator $\epsilon_{\mathcal{D}}(f)$ is the sample average, estimating the population error is simply the classic problem of mean estimation, which you may recall from Section 2.6.

An important classical result from probability theory called the *central limit theorem* guarantees that whenever we possess n random samples $a_1, ..., a_n$ drawn from any distribution with mean μ and standard deviation σ, then, as the number of samples n approaches infinity, the sample average $\hat{\mu}$ approximately tends towards a normal distribution centered at the true mean and with standard deviation $\sigma/\sqrt{n}$. Already, this tells us something important: as the number of examples grows large, our test error $\epsilon_{\mathcal{D}}(f)$ should approach the true error $\epsilon(f)$ at a rate of $O(1/\sqrt{n})$. Thus, to estimate our test error twice as precisely, we must collect four times as large a test set. To reduce our test error by a factor of one hundred, we must collect ten thousand times as large a test set. In general, such a rate of $O(1/\sqrt{n})$ is often the best we can hope for in statistics.

Now that we know something about the asymptotic rate at which our test error $\epsilon_{\mathcal{D}}(f)$ converges to the true error $\epsilon(f)$, we can zoom in on some important details. Recall that the random variable of interest $\mathbf{1}(f(X) \neq Y)$ can only take values 0 and 1 and thus is a Bernoulli random variable, characterized by a parameter indicating the probability that it takes value 1. Here, 1 means that our classifier made an error, so the parameter of our random variable is actually the true error rate $\epsilon(f)$. The variance σ^2 of a Bernoulli depends on its parameter (here, $\epsilon(f)$) according to the expression $\epsilon(f)(1 - \epsilon(f))$. While $\epsilon(f)$ is initially unknown, we know that it cannot be greater than 1. A little investigation of this function reveals that our variance is highest when the true error rate is close to 0.5 and can be far lower when it is close to 0 or close to 1. This tells us that the asymptotic standard deviation of our estimate $\epsilon_{\mathcal{D}}(f)$ of the error $\epsilon(f)$ (over the choice of the n test samples) cannot be any greater than $\sqrt{0.25/n}$.

If we ignore the fact that this rate characterizes behavior as the test set size approaches infinity rather than when we possess finite samples, this tells us that if we want our test error $\epsilon_{\mathcal{D}}(f)$ to approximate the population error $\epsilon(f)$ such that one standard deviation corresponds to an interval of ± 0.01, then we should collect roughly 2500 samples. If we want to fit two standard deviations in that range and thus be 95% confident that $\epsilon_{\mathcal{D}}(f) \in \epsilon(f) \pm 0.01$, then we will need 10,000 samples!

This turns out to be the size of the test sets for many popular benchmarks in machine learning. You might be surprised to find out that thousands of applied deep learning papers get published every year making a big deal out of error rate improvements of 0.01 or less. Of course, when the error rates are much closer to 0, then an improvement of 0.01 can indeed be a big deal.

One pesky feature of our analysis thus far is that it really only tells us about asymptotics, i.e., how the relationship between $\epsilon_\mathcal{D}$ and ϵ evolves as our sample size goes to infinity. Fortunately, because our random variable is bounded, we can obtain valid finite sample bounds by applying an inequality due to Hoeffding (1963):

$$P(\epsilon_\mathcal{D}(f) - \epsilon(f) \geq t) < \exp\left(-2nt^2\right). \tag{4.6.3}$$

Solving for the smallest dataset size that would allow us to conclude with 95% confidence that the distance t between our estimate $\epsilon_\mathcal{D}(f)$ and the true error rate $\epsilon(f)$ does not exceed 0.01, you will find that roughly 15,000 examples are required as compared to the 10,000 examples suggested by the asymptotic analysis above. If you go deeper into statistics you will find that this trend holds generally. Guarantees that hold even in finite samples are typically slightly more conservative. Note that in the scheme of things, these numbers are not so far apart, reflecting the general usefulness of asymptotic analysis for giving us ballpark figures even if they are not guarantees we can take to court.

4.6.2 Test Set Reuse

In some sense, you are now set up to succeed at conducting empirical machine learning research. Nearly all practical models are developed and validated based on test set performance and you are now a master of the test set. For any fixed classifier f, you know how to evaluate its test error $\epsilon_\mathcal{D}(f)$, and know precisely what can (and cannot) be said about its population error $\epsilon(f)$.

So let's say that you take this knowledge and prepare to train your first model f_1. Knowing just how confident you need to be in the performance of your classifier's error rate you apply our analysis above to determine an appropriate number of examples to set aside for the test set. Moreover, let's assume that you took the lessons from Section 3.6 to heart and made sure to preserve the sanctity of the test set by conducting all of your preliminary analysis, hyperparameter tuning, and even selection among multiple competing model architectures on a validation set. Finally you evaluate your model f_1 on the test set and report an unbiased estimate of the population error with an associated confidence interval.

So far everything seems to be going well. However, that night you wake up at 3am with a brilliant idea for a new modeling approach. The next day, you code up your new model, tune its hyperparameters on the validation set and not only are you getting your new model f_2 to work but its error rate appears to be much lower than f_1's. However, the thrill of discovery suddenly fades as you prepare for the final evaluation. You do not have a test set!

Even though the original test set $\mathcal{D}$ is still sitting on your server, you now face two formidable problems. First, when you collected your test set, you determined the required level of precision under the assumption that you were evaluating a single classifier f. However, if you get into the business of evaluating multiple classifiers $f_1, ..., f_k$ on the same test set, you must consider the problem of false discovery. Before, you might have been 95% sure that $\epsilon_{\mathcal{D}}(f) \in \epsilon(f) \pm 0.01$ for a single classifier f and thus the probability of a misleading result was a mere 5%. With k classifiers in the mix, it can be hard to guarantee that there is not even one among them whose test set performance is misleading. With 20 classifiers under consideration, you might have no power at all to rule out the possibility that at least one among them received a misleading score. This problem relates to multiple hypothesis testing, which despite a vast literature in statistics, remains a persistent problem plaguing scientific research.

If that is not enough to worry you, there is a special reason to distrust the results that you get on subsequent evaluations. Recall that our analysis of test set performance rested on the assumption that the classifier was chosen absent any contact with the test set and thus we could view the test set as drawn randomly from the underlying population. Here, not only are you testing multiple functions, the subsequent function f_2 was chosen after you observed the test set performance of f_1. Once information from the test set has leaked to the modeler, it can never be a true test set again in the strictest sense. This problem is called *adaptive overfitting* and has recently emerged as a topic of intense interest to learning theorists and statisticians (Dwork *et al.*, 2015). Fortunately, while it is possible to leak all information out of a holdout set, and the theoretical worst case scenarios are bleak, these analyses may be too conservative. In practice, take care to create real test sets, to consult them as infrequently as possible, to account for multiple hypothesis testing when reporting confidence intervals, and to dial up your vigilance more aggressively when the stakes are high and your dataset size is small. When running a series of benchmark challenges, it is often good practice to maintain several test sets so that after each round, the old test set can be demoted to a validation set.

4.6.3 Statistical Learning Theory

Put simply, *test sets are all that we really have*, and yet this fact seems strangely unsatisfying. First, we seldom possess a *true test set*—unless we are the ones creating the dataset, someone else has probably already evaluated their own classifier on our ostensible "test set". And even when we have first dibs, we soon find ourselves frustrated, wishing we could evaluate our subsequent modeling attempts without the gnawing feeling that we cannot trust our numbers. Moreover, even a true test set can only tell us *post hoc* whether a classifier has in fact generalized to the population, not whether we have any reason to expect *a priori* that it should generalize.

With these misgivings in mind, you might now be sufficiently primed to see the appeal of *statistical learning theory*, the mathematical subfield of machine learning whose practitioners aim to elucidate the fundamental principles that explain why/when models trained on empirical data can/will generalize to unseen data. One of the primary aims of statistical learning researchers has been to bound the generalization gap, relating the properties of the model class to the number of samples in the dataset.

Learning theorists aim to bound the difference between the *empirical error* $\epsilon_{\mathcal{S}}(f_{\mathcal{S}})$ of a learned classifier $f_{\mathcal{S}}$, both trained and evaluated on the training set $\mathcal{S}$, and the true error $\epsilon(f_{\mathcal{S}})$ of that same classifier on the underlying population. This might look similar to the evaluation problem that we just addressed but there is a major difference. Earlier, the classifier f was fixed and we only needed a dataset for evaluative purposes. And indeed, any fixed classifier does generalize: its error on a (previously unseen) dataset is an unbiased estimate of the population error. But what can we say when a classifier is trained and evaluated on the same dataset? Can we ever be confident that the training error will be close to the testing error?

Suppose that our learned classifier f_S must be chosen from some pre-specified set of functions $\mathcal{F}$. Recall from our discussion of test sets that while it is easy to estimate the error of a single classifier, things get hairy when we begin to consider collections of classifiers. Even if the empirical error of any one (fixed) classifier will be close to its true error with high probability, once we consider a collection of classifiers, we need to worry about the possibility that *just one* of them will receive a badly estimated error. The worry is that we might pick such a classifier and thereby grossly underestimate the population error. Moreover, even for linear models, because their parameters are continuously valued, we are typically choosing from an infinite class of functions ($|\mathcal{F}| = \infty$).

One ambitious solution to the problem is to develop analytic tools for proving uniform convergence, i.e., that with high probability, the empirical error rate for every classifier in the class $f \in \mathcal{F}$ will *simultaneously* converge to its true error rate. In other words, we seek a theoretical principle that would allow us to state that with probability at least $1 - \delta$ (for some small δ) no classifier's error rate $\epsilon(f)$ (among all classifiers in the class $\mathcal{F}$) will be misestimated by more than some small amount α. Clearly, we cannot make such statements for all model classes $\mathcal{F}$. Recall the class of memorization machines that always achieve empirical error 0 but never outperform random guessing on the underlying population.

In a sense the class of memorizers is too flexible. No such a uniform convergence result could possibly hold. On the other hand, a fixed classifier is useless—it generalizes perfectly, but fits neither the training data nor the test data. The central question of learning has thus historically been framed as a trade-off between more flexible (higher variance) model classes that better fit the training data but risk overfitting, versus more rigid (higher bias) model classes that generalize well but risk underfitting. A central question in learning theory has been to develop the appropriate mathematical analysis to quantify where a model sits along this spectrum, and to provide the associated guarantees.

In a series of seminal papers, Vapnik and Chervonenkis extended the theory on the convergence of relative frequencies to more general classes of functions (Vapnik and Chervonenkis, 1964, Vapnik and Chervonenkis, 1968, Vapnik and Chervonenkis, 1971, Vapnik and Chervonenkis, 1981, Vapnik and Chervonenkis, 1991, Vapnik and Chervonenkis, 1974). One of the key contributions of this line of work is the Vapnik–Chervonenkis (VC) dimension, which measures (one notion of) the complexity (flexibility) of a model class. Moreover, one of their key results bounds the difference between the empirical error and the population error as a function of the VC dimension and the number of samples:

$$P\left(R[p, f] - R_{\textrm{emp}}[\mathbf{X}, \mathbf{Y}, f] < \alpha\right) \geq 1 - \delta \ \textrm{ for } \ \alpha \geq c\sqrt{(\textrm{VC} - \log \delta)/n}. \tag{4.6.4}$$

Here $\delta > 0$ is the probability that the bound is violated, α is the upper bound on the generalization gap, and n is the dataset size. Lastly, $c > 0$ is a constant that depends only on the scale of the loss that can be incurred. One use of the bound might be to plug in desired values of δ and α to determine how many samples to collect. The VC dimension quantifies the largest number of data points for which we can assign any arbitrary (binary) labeling and for each find some model f in the class that agrees with that labeling. For example, linear models on d-dimensional inputs have VC dimension $d + 1$. It is easy to see that a line can assign any possible labeling to three points in two dimensions, but not to four. Unfortunately, the theory tends to be overly pessimistic for more complex models and obtaining this guarantee typically requires far more examples than are actually needed to achieve the desired error rate. Note also that fixing the model class and δ, our error rate again decays with the usual $O(1/\sqrt{n})$ rate. It seems unlikely that we could do better in terms of n. However, as we vary the model class, VC dimension can present a pessimistic picture of the generalization gap.

4.6.4 Summary

The most straightforward way to evaluate a model is to consult a test set comprised of previously unseen data. Test set evaluations provide an unbiased estimate of the true error and converge at the desired $O(1/\sqrt{n})$ rate as the test set grows. We can provide approximate confidence intervals based on exact asymptotic distributions or valid finite sample confidence intervals based on (more conservative) finite sample guarantees. Indeed test set evaluation is the bedrock of modern machine learning research. However, test sets are seldom true test sets (used by multiple researchers again and again). Once the same test set is used to evaluate multiple models, controlling for false discovery can be difficult. This can cause huge problems in theory. In practice, the significance of the problem depends on the size of the holdout sets in question and whether they are merely being used to choose hyperparameters or if they are leaking information more directly. Nevertheless, it is good practice to curate real test sets (or multiple) and to be as conservative as possible about how often they are used.

Hoping to provide a more satisfying solution, statistical learning theorists have developed methods for guaranteeing uniform convergence over a model class. If indeed every model's empirical error simultaneously converges to its true error, then we are free to choose the model that performs best, minimizing the training error, knowing that it too will perform similarly well on the holdout data. Crucially, any one of such results must depend on some property of the model class. Vladimir Vapnik and Alexey Chernovenkis introduced the VC dimension, presenting uniform convergence results that hold for all models in a VC class. The training errors for all models in the class are (simultaneously) guaranteed to be close to their true errors, and guaranteed to grow even closer at $O(1/\sqrt{n})$ rates. Following the revolutionary discovery of VC dimension, numerous alternative complexity measures have been proposed, each facilitating an analogous generalization guarantee. See Boucheron *et al.* (2005) for a detailed discussion of several advanced ways of measuring function complexity. Unfortunately, while these complexity measures have become broadly useful tools in statistical theory, they turn out to be powerless (as straightforwardly applied) for explaining why deep neural networks generalize. Deep neural networks often have millions of parameters (or more), and can easily assign random labels to large collections of points. Nevertheless, they generalize well on practical problems and, surprisingly, they often generalize better, when they are larger and deeper, despite incurring higher VC dimensions. In the next chapter, we will revisit generalization in the context of deep learning.

4.6.5 Exercises

1. If we wish to estimate the error of a fixed model f to within 0.0001 with probability greater than 99.9%, how many samples do we need?
2. Suppose that somebody else possesses a labeled test set $\mathcal{D}$ and only makes available the unlabeled inputs (features). Now suppose that you can only access the test set labels by running a model f (with no restrictions placed on the model class) on each of the unlabeled inputs and receiving the corresponding error $\epsilon_{\mathcal{D}}(f)$. How many models would you need to evaluate before you leak the entire test set and thus could appear to have error 0, regardless of your true error?
3. What is the VC dimension of the class of fifth-order polynomials?
4. What is the VC dimension of axis-aligned rectangles on two-dimensional data?

100

Discussions[100].

4.7 Environment and Distribution Shift

In the previous sections, we worked through a number of hands-on applications of machine learning, fitting models to a variety of datasets. And yet, we never stopped to contemplate either where data came from in the first place or what we ultimately plan to do with the outputs from our models. Too often, machine learning developers in possession of data rush to develop models without pausing to consider these fundamental issues.

Many failed machine learning deployments can be traced back to this failure. Sometimes models appear to perform marvelously as measured by test set accuracy but fail catastrophically in deployment when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst that perturbs the data distribution. Say, for example, that we trained a model to predict who will repay rather than default on a loan, finding that an applicant's choice of footwear was associated with the risk of default (Oxfords indicate repayment, sneakers indicate default). We might be inclined thereafter to grant a loan to any applicant wearing Oxfords and to deny all applicants wearing sneakers.

In this case, our ill-considered leap from pattern recognition to decision-making and our failure to critically consider the environment might have disastrous consequences. For starters, as soon as we began making decisions based on footwear, customers would catch on and change their behavior. Before long, all applicants would be wearing Oxfords, without any coincident improvement in credit-worthiness. Take a minute to digest this because similar issues abound in many applications of machine learning: by introducing our model-based decisions to the environment, we might break the model.

While we cannot possibly give these topics a complete treatment in one section, we aim here to expose some common concerns, and to stimulate the critical thinking required to detect such situations early, mitigate damage, and use machine learning responsibly. Some of the solutions are simple (ask for the "right" data), some are technically difficult (implement a reinforcement learning system), and others require that we step outside the realm of statistical prediction altogether and grapple with difficult philosophical questions concerning the ethical application of algorithms.

4.7.1 Types of Distribution Shift

To begin, we stick with the passive prediction setting considering the various ways that data distributions might shift and what might be done to salvage model performance. In one classic setup, we assume that our training data was sampled from some distribution $p_S(\mathbf{x}, y)$ but that our test data will consist of unlabeled examples drawn from some different distribution $p_T(\mathbf{x}, y)$. Already, we must confront a sobering reality. Absent any assumptions on how p_S and p_T relate to each other, learning a robust classifier is impossible.

Consider a binary classification problem, where we wish to distinguish between dogs and cats. If the distribution can shift in arbitrary ways, then our setup permits the pathological case in which the distribution over inputs remains constant: $p_S(\mathbf{x}) = p_T(\mathbf{x})$, but the labels are all flipped: $p_S(y \mid \mathbf{x}) = 1 - p_T(y \mid \mathbf{x})$. In other words, if God can suddenly decide that in the future all "cats" are now dogs and what we previously called "dogs" are now cats—without any change in the distribution of inputs $p(\mathbf{x})$, then we cannot possibly distinguish this setting from one in which the distribution did not change at all.

Fortunately, under some restricted assumptions on the ways our data might change in the future, principled algorithms can detect shift and sometimes even adapt on the fly, improving on the accuracy of the original classifier.

Covariate Shift

Among categories of distribution shift, covariate shift may be the most widely studied. Here, we assume that while the distribution of inputs may change over time, the labeling function, i.e., the conditional distribution $P(y \mid \mathbf{x})$ does not change. Statisticians call this *covariate shift* because the problem arises due to a shift in the distribution of the covariates (features). While we can sometimes reason about distribution shift without invoking causality, we note that covariate shift is the natural assumption to invoke in settings where we believe that $\mathbf{x}$ causes y.

Consider the challenge of distinguishing cats and dogs. Our training data might consist of images of the kind in Fig. 4.7.1.

At test time we are asked to classify the images in Fig. 4.7.2.

The training set consists of photos, while the test set contains only cartoons. Training on a

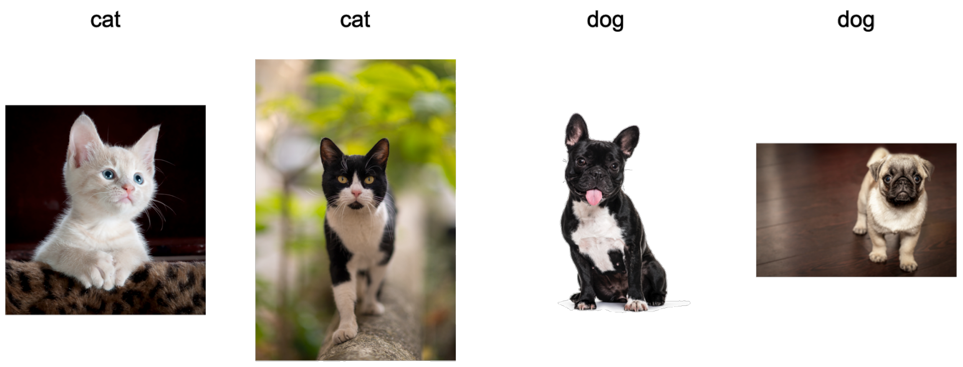


Fig. 4.7.1 Training data for distinguishing cats and dogs (illustrations: Lafeez Hossain / 500px / Getty Images; ilkermetinkursova / iStock / Getty Images Plus; GlobalP / iStock / Getty Images Plus; Musthafa Aboobakuru / 500px / Getty Images).

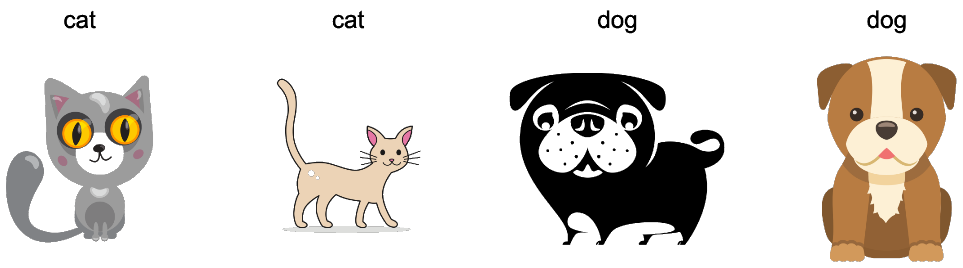


Fig. 4.7.2 Test data for distinguishing cats and dogs (illustrations: SIBAS_minich / iStock / Getty Images Plus; Ghrzuzudu / iStock / Getty Images Plus; id-work / DigitalVision Vectors / Getty Images; Yime / iStock / Getty Images Plus).

dataset with substantially different characteristics from the test set can spell trouble absent a coherent plan for how to adapt to the new domain.

Label Shift

Label shift describes the converse problem. Here, we assume that the label marginal $P(y)$ can change but the class-conditional distribution $P(\mathbf{x} \mid y)$ remains fixed across domains. Label shift is a reasonable assumption to make when we believe that y causes $\mathbf{x}$. For example, we may want to predict diagnoses given their symptoms (or other manifestations), even as the relative prevalence of diagnoses are changing over time. Label shift is the appropriate assumption here because diseases cause symptoms. In some degenerate cases the label shift and covariate shift assumptions can hold simultaneously. For example, when the label is deterministic, the covariate shift assumption will be satisfied, even when y causes $\mathbf{x}$. Interestingly, in these cases, it is often advantageous to work with methods that flow from the label shift assumption. That is because these methods tend to involve manipulating objects that look like labels (often low-dimensional), as opposed to objects that look like inputs, which tend to be high-dimensional in deep learning.

Concept Shift

We may also encounter the related problem of *concept shift*, which arises when the very definitions of labels can change. This sounds weird—a *cat* is a *cat*, no? However, other categories are subject to changes in usage over time. Diagnostic criteria for mental illness, what passes for fashionable, and job titles, are all subject to considerable amounts of concept shift. It turns out that if we navigate around the United States, shifting the source of our data by geography, we will find considerable concept shift regarding the distribution of names for *soft drinks* as shown in Fig. 4.7.3.

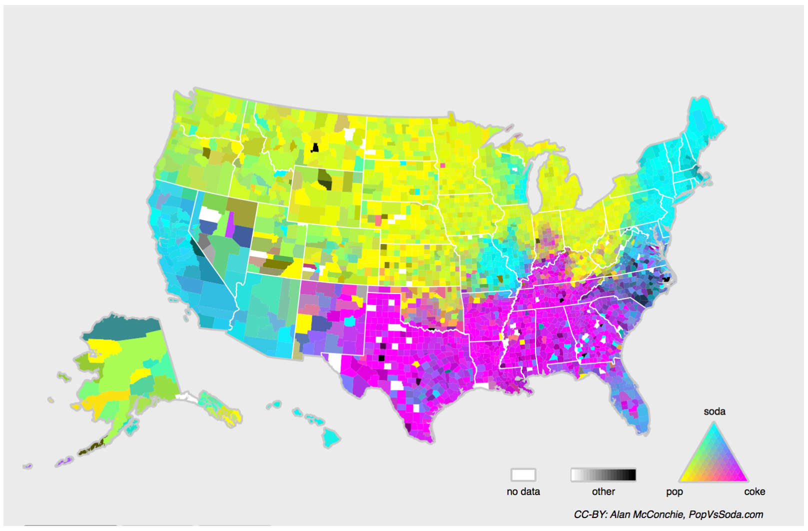


Fig. 4.7.3 Concept shift for soft drink names in the United States (CC-BY: Alan McConchie, PopVsSoda.com).

If we were to build a machine translation system, the distribution $P(y \mid \mathbf{x})$ might be different depending on our location. This problem can be tricky to spot. We might hope to exploit knowledge that shift only takes place gradually either in a temporal or geographic sense.

4.7.2 Examples of Distribution Shift

Before delving into formalism and algorithms, we can discuss some concrete situations where covariate or concept shift might not be obvious.

Medical Diagnostics

Imagine that you want to design an algorithm to detect cancer. You collect data from healthy and sick people and you train your algorithm. It works fine, giving you high accuracy and you conclude that you are ready for a successful career in medical diagnostics. *Not so fast.*

The distributions that gave rise to the training data and those you will encounter in the wild might differ considerably. This happened to an unfortunate startup that some of we authors worked with years ago. They were developing a blood test for a disease that predominantly affects older men and hoped to study it using blood samples that they had collected from patients. However, it is considerably more difficult to obtain blood samples from healthy men than from sick patients already in the system. To compensate, the startup solicited blood donations from students on a university campus to serve as healthy controls in developing their test. Then they asked whether we could help them to build a classifier for detecting the disease.

As we explained to them, it would indeed be easy to distinguish between the healthy and sick cohorts with near-perfect accuracy. However, that is because the test subjects differed in age, hormone levels, physical activity, diet, alcohol consumption, and many more factors unrelated to the disease. This was unlikely to be the case with real patients. Due to their sampling procedure, we could expect to encounter extreme covariate shift. Moreover, this case was unlikely to be correctable via conventional methods. In short, they wasted a significant sum of money.

Self-Driving Cars

Say a company wanted to leverage machine learning for developing self-driving cars. One key component here is a roadside detector. Since real annotated data is expensive to get, they had the (smart and questionable) idea to use synthetic data from a game rendering engine as additional training data. This worked really well on "test data" drawn from the rendering engine. Alas, inside a real car it was a disaster. As it turned out, the roadside had been rendered with a very simplistic texture. More importantly, *all* the roadside had been rendered with the *same* texture and the roadside detector learned about this "feature" very quickly.

A similar thing happened to the US Army when they first tried to detect tanks in the forest. They took aerial photographs of the forest without tanks, then drove the tanks into the forest and took another set of pictures. The classifier appeared to work *perfectly*. Unfortunately, it had merely learned how to distinguish trees with shadows from trees without shadows—the first set of pictures was taken in the early morning, the second set at noon.

Nonstationary Distributions

A much more subtle situation arises when the distribution changes slowly (also known as *nonstationary distribution*) and the model is not updated adequately. Below are some typical cases.

- We train a computational advertising model and then fail to update it frequently (e.g., we forget to incorporate that an obscure new device called an iPad was just launched).
- We build a spam filter. It works well at detecting all spam that we have seen so far. But then the spammers wise up and craft new messages that look unlike anything we have seen before.

- We build a product recommendation system. It works throughout the winter but then continues to recommend Santa hats long after Christmas.

More Anecdotes

- We build a face detector. It works well on all benchmarks. Unfortunately it fails on test data—the offending examples are close-ups where the face fills the entire image (no such data was in the training set).
- We build a web search engine for the US market and want to deploy it in the UK.
- We train an image classifier by compiling a large dataset where each among a large set of classes is equally represented in the dataset, say 1000 categories, represented by 1000 images each. Then we deploy the system in the real world, where the actual label distribution of photographs is decidedly non-uniform.

4.7.3 Correction of Distribution Shift

As we have discussed, there are many cases where training and test distributions $P(\mathbf{x}, y)$ are different. In some cases, we get lucky and the models work despite covariate, label, or concept shift. In other cases, we can do better by employing principled strategies to cope with the shift. The remainder of this section grows considerably more technical. The impatient reader could continue on to the next section as this material is not prerequisite to subsequent concepts.

Empirical Risk and Risk

Let's first reflect on what exactly is happening during model training: we iterate over features and associated labels of training data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and update the parameters of a model f after every minibatch. For simplicity we do not consider regularization, so we largely minimize the loss on the training:

$$\underset{f}{\text{minimize}}\, \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i), \tag{4.7.1}$$

where l is the loss function measuring "how bad" the prediction $f(\mathbf{x}_i)$ is given the associated label y_i. Statisticians call the term in (4.7.1) *empirical risk*. The *empirical risk* is an average loss over the training data for approximating the *risk*, which is the expectation of the loss over the entire population of data drawn from their true distribution $p(\mathbf{x}, y)$:

$$E_{p(\mathbf{x}, y)}[l(f(\mathbf{x}), y)] = \int\int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \;d\mathbf{x}dy. \tag{4.7.2}$$

However, in practice we typically cannot obtain the entire population of data. Thus, *empirical risk minimization*, which is minimizing the empirical risk in (4.7.1), is a practical strategy for machine learning, with the hope of approximately minimizing the risk.

Covariate Shift Correction

Assume that we want to estimate some dependency $P(y \mid \mathbf{x})$ for which we have labeled data $(\mathbf{x}_i, y_i)$. Unfortunately, the observations $\mathbf{x}_i$ are drawn from some *source distribution* $q(\mathbf{x})$ rather than the *target distribution* $p(\mathbf{x})$. Fortunately, the dependency assumption means that the conditional distribution does not change: $p(y \mid \mathbf{x}) = q(y \mid \mathbf{x})$. If the source distribution $q(\mathbf{x})$ is "wrong", we can correct for that by using the following simple identity in the risk:

$$\int\int l(f(\mathbf{x}), y)p(y \mid \mathbf{x})p(\mathbf{x}) \;d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y)q(y \mid \mathbf{x})q(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})} \;d\mathbf{x}dy. \tag{4.7.3}$$

In other words, we need to reweigh each data example by the ratio of the probability that it would have been drawn from the correct distribution to that from the wrong one:

$$\beta_i \stackrel{\textrm{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{4.7.4}$$

Plugging in the weight β_i for each data example $(\mathbf{x}_i, y_i)$ we can train our model using *weighted empirical risk minimization*:

$$\mathop{\textrm{minimize}}_f \frac{1}{n} \sum_{i=1}^n \beta_i l(f(\mathbf{x}_i), y_i). \tag{4.7.5}$$

Alas, we do not know that ratio, so before we can do anything useful we need to estimate it. Many methods are available, including some fancy operator-theoretic approaches that attempt to recalibrate the expectation operator directly using a minimum-norm or a maximum entropy principle. Note that for any such approach, we need samples drawn from both distributions—the "true" p, e.g., by access to test data, and the one used for generating the training set q (the latter is trivially available). Note however, that we only need features $\mathbf{x} \sim p(\mathbf{x})$; we do not need to access labels $y \sim p(y)$.

In this case, there exists a very effective approach that will give almost as good results as the original: namely, logistic regression, which is a special case of softmax regression (see Section 4.1) for binary classification. This is all that is needed to compute estimated probability ratios. We learn a classifier to distinguish between data drawn from $p(\mathbf{x})$ and data drawn from $q(\mathbf{x})$. If it is impossible to distinguish between the two distributions then it means that the associated instances are equally likely to come from either one of those two distributions. On the other hand, any instances that can be well discriminated should be significantly overweighted or underweighted accordingly.

For simplicity's sake assume that we have an equal number of instances from both distributions $p(\mathbf{x})$ and $q(\mathbf{x})$, respectively. Now denote by z labels that are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$P(z=1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x})+q(\mathbf{x})} \textrm{ and hence } \frac{P(z=1 \mid \mathbf{x})}{P(z=-1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{4.7.6}$$

Thus, if we use a logistic regression approach, where $P(z=1 \mid \mathbf{x})=\frac{1}{1+\exp(-h(\mathbf{x}))}$ (h is a parametrized function), it follows that

$$\beta_i = \frac{1/(1+\exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1+\exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{4.7.7}$$

As a result, we need to solve two problems: the first, to distinguish between data drawn from both distributions, and then a weighted empirical risk minimization problem in (4.7.5) where we weigh terms by β_i.

Now we are ready to describe a correction algorithm. Suppose that we have a training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and an unlabeled test set $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$. For covariate shift, we assume that $\mathbf{x}_i$ for all $1 \le i \le n$ are drawn from some source distribution and $\mathbf{u}_i$ for all $1 \le i \le m$ are drawn from the target distribution. Here is a prototypical algorithm for correcting covariate shift:

1. Create a binary-classification training set: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. Train a binary classifier using logistic regression to get the function h.
3. Weigh training data using $\beta_i = \exp(h(\mathbf{x}_i))$ or better $\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$ for some constant c.
4. Use weights β_i for training on $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ in (4.7.5).

Note that the above algorithm relies on a crucial assumption. For this scheme to work, we need that each data example in the target (e.g., test time) distribution had nonzero probability of occurring at training time. If we find a point where $p(\mathbf{x}) > 0$ but $q(\mathbf{x}) = 0$, then the corresponding importance weight should be infinity.

Label Shift Correction

Assume that we are dealing with a classification task with k categories. Using the same notation in Section 4.7.3, q and p are the source distribution (e.g., training time) and target distribution (e.g., test time), respectively. Assume that the distribution of labels shifts over time: $q(y) \neq p(y)$, but the class-conditional distribution stays the same: $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$. If the source distribution $q(y)$ is "wrong", we can correct for that according to the following identity in the risk as defined in (4.7.2):

$$\int\int l(f(\mathbf{x}), y) p(\mathbf{x} \mid y) p(y) \; d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y) q(\mathbf{x} \mid y) q(y) \frac{p(y)}{q(y)} \; d\mathbf{x}dy. \tag{4.7.8}$$

Here, our importance weights will correspond to the label likelihood ratios:

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{4.7.9}$$

One nice thing about label shift is that if we have a reasonably good model on the source distribution, then we can get consistent estimates of these weights without ever having to deal with the ambient dimension. In deep learning, the inputs tend to be high-dimensional objects like images, while the labels are often simpler objects like categories.

To estimate the target label distribution, we first take our reasonably good off-the-shelf classifier (typically trained on the training data) and compute its "confusion" matrix using the validation set (also from the training distribution). The *confusion matrix*, $\mathbf{C}$, is simply a $k \times k$ matrix, where each column corresponds to the label category (ground truth) and each row corresponds to our model's predicted category. Each cell's value c_{ij} is the fraction of total predictions on the validation set where the true label was j and our model predicted i.

Now, we cannot calculate the confusion matrix on the target data directly because we do not get to see the labels for the examples that we see in the wild, unless we invest in a complex real-time annotation pipeline. What we can do, however, is average all of our model's predictions at test time together, yielding the mean model outputs $\mu(\hat{\mathbf{y}}) \in \mathbb{R}^k$, where the $i^\textrm{th}$ element $\mu(\hat{y}_i)$ is the fraction of the total predictions on the test set where our model predicted i.

It turns out that under some mild conditions—if our classifier was reasonably accurate in the first place, and if the target data contains only categories that we have seen before, and if the label shift assumption holds in the first place (the strongest assumption here)—we can estimate the test set label distribution by solving a simple linear system

$$\mathbf{C} p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{4.7.10}$$

because as an estimate $\sum_{j=1}^k c_{ij} p(y_j) = \mu(\hat{y}_i)$ holds for all $1 \leq i \leq k$, where $p(y_j)$ is the $j^\textrm{th}$ element of the k-dimensional label distribution vector $p(\mathbf{y})$. If our classifier is sufficiently accurate to begin with, then the confusion matrix $\mathbf{C}$ will be invertible, and we get a solution $p(\mathbf{y}) = \mathbf{C}^{-1} \mu(\hat{\mathbf{y}})$.

Because we observe the labels on the source data, it is easy to estimate the distribution $q(y)$. Then, for any training example i with label y_i, we can take the ratio of our estimated $p(y_i)/q(y_i)$ to calculate the weight β_i, and plug this into weighted empirical risk minimization in (4.7.5).

Concept Shift Correction

Concept shift is much harder to fix in a principled manner. For instance, in a situation where suddenly the problem changes from distinguishing cats from dogs to one of distinguishing white from black animals, it will be unreasonable to assume that we can do much better than just collecting new labels and training from scratch. Fortunately, in practice, such extreme shifts are rare. Instead, what usually happens is that the task keeps on changing slowly. To make things more concrete, here are some examples:

- In computational advertising, new products are launched, old products become less popular. This means that the distribution over ads and their popularity changes gradually and any click-through rate predictor needs to change gradually with it.
- Traffic camera lenses degrade gradually due to environmental wear, affecting image quality progressively.
- News content changes gradually (i.e., most of the news remains unchanged but new stories appear).

In such cases, we can use the same approach that we used for training networks to make them adapt to the change in the data. In other words, we use the existing network weights and simply perform a few update steps with the new data rather than training from scratch.

4.7.4 A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in distributions, we can now consider some other aspects of machine learning problem formulation.

Batch Learning

In *batch learning*, we have access to training features and labels $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, which we use to train a model $f(\mathbf{x})$. Later on, we deploy this model to score new data $(\mathbf{x}, y)$ drawn from the same distribution. This is the default assumption for any of the problems that we discuss here. For instance, we might train a cat detector based on lots of pictures of cats and dogs. Once we have trained it, we ship it as part of a smart catdoor computer vision system that lets only cats in. This is then installed in a customer's home and is never updated again (barring extreme circumstances).

Online Learning

Now imagine that the data $(\mathbf{x}_i, y_i)$ arrives one sample at a time. More specifically, assume that we first observe $\mathbf{x}_i$, then we need to come up with an estimate $f(\mathbf{x}_i)$. Only once we have done this do we observe y_i and so receive a reward or incur a loss, given our decision. Many real problems fall into this category. For example, we need to predict tomorrow's stock price, which allows us to trade based on that estimate and at the end of the day we find out whether our estimate made us a profit. In other words, in *online learning*, we have the following cycle where we are continuously improving our model given new observations:

$$\begin{aligned} &\text{model } f_t \longrightarrow \text{data } \mathbf{x}_t \longrightarrow \text{estimate } f_t(\mathbf{x}_t) \longrightarrow \\ &\text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \text{model } f_{t+1} \end{aligned} \tag{4.7.11}$$

Bandits

Bandits are a special case of the problem above. While in most learning problems we have a continuously parametrized function f where we want to learn its parameters (e.g., a deep network), in a *bandit* problem we only have a finite number of arms that we can pull, i.e., a finite number of actions that we can take. It is not very surprising that for this simpler problem stronger theoretical guarantees in terms of optimality can be obtained. We list it mainly since this problem is often (confusingly) treated as if it were a distinct learning setting.

Control

In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it will just remember and the response will depend on what happened before. For instance, a coffee boiler controller will observe different temperatures depending on whether it was heating the boiler previously. PID (proportional-integral-derivative) controller algorithms are a popular choice there. Likewise, a user's behavior on a news site will depend on what we showed them previously (e.g., they will read most news only once). Many such algorithms form a model of the environment in which they act so as to make their decisions appear less random. Recently, control theory (e.g., PID variants) has also been used to automatically tune hyperparameters to achieve better disentangling and reconstruction quality, and improve the diversity of generated text and the reconstruction quality of generated images (Shao *et al.*, 2020).

Reinforcement Learning

In the more general case of an environment with memory, we may encounter situations where the environment is trying to cooperate with us (cooperative games, in particular for non-zero-sum games), or others where the environment will try to win. Chess, Go, Backgammon, or StarCraft are some of the cases in *reinforcement learning*. Likewise, we might want to build a good controller for autonomous cars. Other cars are likely to respond to the autonomous car's driving style in nontrivial ways, e.g., trying to avoid it, trying to cause an accident, or trying to cooperate with it.

Considering the Environment

One key distinction between the different situations above is that a strategy that might have worked throughout in the case of a stationary environment, might not work throughout in an environment that can adapt. For instance, an arbitrage opportunity discovered by a trader is likely to disappear once it is exploited. The speed and manner at which the environment changes determines to a large extent the type of algorithms that we can bring to bear. For instance, if we know that things may only change slowly, we can force any estimate to change only slowly, too. If we know that the environment might change instantaneously, but only very infrequently, we can make allowances for that. These types of knowledge are crucial for the aspiring data scientist in dealing with concept shift, i.e., when the problem that is being solved can change over time.

4.7.5 Fairness, Accountability, and Transparency in Machine Learning

Finally, it is important to remember that when you deploy machine learning systems you are not merely optimizing a predictive model—you are typically providing a tool that will be used to (partially or fully) automate decisions. These technical systems can impact the lives of individuals who are subject to the resulting decisions. The leap from considering predictions to making decisions raises not only new technical questions, but also a slew of ethical questions that must be carefully considered. If we are deploying a medical diagnostic system, we need to know for which populations it may work and for which it may not. Overlooking foreseeable risks to the welfare of a subpopulation could cause us to administer inferior care. Moreover, once we contemplate decision-making systems, we must step back and reconsider how we evaluate our technology. Among other consequences of this change of scope, we will find that *accuracy* is seldom the right measure. For instance, when translating predictions into actions, we will often want to take into account the potential cost sensitivity of erring in various ways. If one way of misclassifying an image could be perceived as a racial sleight of hand, while misclassification to a different category would be harmless, then we might want to adjust our thresholds accordingly, accounting for societal values in designing the decision-making protocol. We also want to be careful about how prediction systems can lead to feedback loops. For example, consider predictive policing systems, which allocate patrol officers to areas with high forecasted crime. It is easy to see how a worrying pattern can emerge:

1. Neighborhoods with more crime get more patrols.
2. Consequently, more crimes are discovered in these neighborhoods, entering the training data available for future iterations.
3. Exposed to more positives, the model predicts yet more crime in these neighborhoods.
4. In the next iteration, the updated model targets the same neighborhood even more heavily leading to yet more crimes discovered, etc.

Often, the various mechanisms by which a model's predictions become coupled to its training data are unaccounted for in the modeling process. This can lead to what researchers call *runaway feedback loops*. Additionally, we want to be careful about whether we are addressing the right problem in the first place. Predictive algorithms now play an outsize role in mediating the dissemination of information. Should the news that an individual encounters be determined by the set of Facebook pages they have *Liked*? These are just a few among the many pressing ethical dilemmas that you might encounter in a career in machine learning.

4.7.6 Summary

In many cases training and test sets do not come from the same distribution. This is called distribution shift. The risk is the expectation of the loss over the entire population of data drawn from their true distribution. However, this entire population is usually unavailable. Empirical risk is an average loss over the training data to approximate the risk. In practice, we perform empirical risk minimization.

Under the corresponding assumptions, covariate and label shift can be detected and corrected for at test time. Failure to account for this bias can become problematic at test time. In some cases, the environment may remember automated actions and respond in surprising ways. We must account for this possibility when building models and continue to monitor live systems, open to the possibility that our models and the environment will become entangled in unanticipated ways.

4.7.7 Exercises

1. What could happen when we change the behavior of a search engine? What might the users do? What about the advertisers?
2. Implement a covariate shift detector. Hint: build a classifier.
3. Implement a covariate shift corrector.
4. Besides distribution shift, what else could affect how the empirical risk approximates the risk?

Discussions[101].

101