4 Multilayer Perceptrons

In this chapter, we will introduce your first truly *deep* networks. The simplest deep networks are called multilayer perceptrons, and they consist of many layers of neurons each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence). When we train high-capacity models we run the risk of overfitting. Thus, we will need to provide your first rigorous introduction to the notions of overfitting, underfitting, and capacity control. To help you combat these problems, we will introduce regularization techniques such as dropout and weight decay. We will also discuss issues relating to numerical stability and parameter initialization that are key to successfully training deep networks. Throughout, we focus on applying models to real data, aiming to give the reader a firm grasp not just of the concepts but also of the practice of using deep networks. We punt matters relating to the computational performance, scalability and efficiency of our models to subsequent chapters.

4.1 Multilayer Perceptron

In the previous chapters, we showed how you could implement multiclass logistic regression (also called softmax regression) for classifying images of clothing into the 10 possible categories. To get there, we had to learn how to wrangle data, coerce our outputs into a valid probability distribution (via `softmax`), how to apply an appropriate loss function, and how to optimize over our parameters. Now that we've covered these preliminaries, we are free to focus our attention on the more exciting enterprise of designing powerful models using deep neural networks.

4.1.1 Hidden Layers

Let's recall linear regression and softmax regression with an example as illustrated in Fig. 4.1.1. In general, we mapped our inputs directly to our outputs via a single linear transformation:

$$\hat{\mathbf{o}} = \mathrm{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{4.1.1}$$

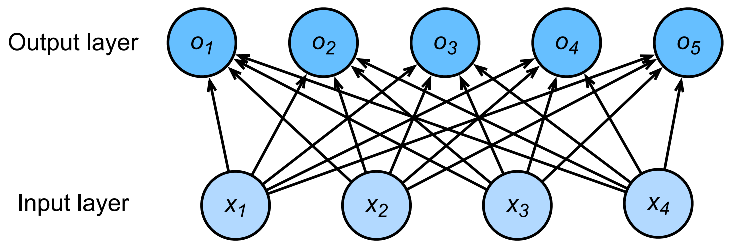


Fig. 4.1.1: Single layer perceptron with 5 output units.

If our labels really were related to our input data by an approximately linear function, then this approach would be perfect. But linearity is a *strong assumption*. Linearity implies that for whatever target value we are trying to predict, increasing the value of each of our inputs should either drive the value of the output up or drive it down, irrespective of the value of the other inputs.

Sometimes this makes sense! Say we are trying to predict whether an individual will or will not repay a loan. We might reasonably imagine that all else being equal, an applicant with a higher income would be more likely to repay than one with a lower income. In these cases, linear models might perform well, and they might even be hard to beat.

But what about classifying images in FashionMNIST? Should increasing the intensity of the pixel at location (13, 17) always increase the likelihood that the image depicts a pocketbook? That seems ridiculous because we all know that you cannot make sense out of an image without accounting for the interactions among pixels.

From One to Many

As another case, consider trying to classify images based on whether they depict *cats* or *dogs* given black-and-white images.

If we use a linear model, we'd basically be saying that for each pixel, increasing its value (making it more white) must always increase the probability that the image depicts a dog or must always increase the probability that the image depicts a cat. We would be making the absurd assumption that the only requirement for differentiating cats vs. dogs is to assess how bright they are. That approach is doomed to fail in a work that contains both black dogs and black cats, and both white dogs and white cats.

Teasing out what is depicted in an image generally requires allowing more complex relationships between our inputs and outputs. Thus we need models capable of discovering patterns that might be characterized by interactions among the many features. We can over come these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers. The easiest way to do this is to stack many layers of neurons on top of each other. Each layer feeds into the layer above it, until we generate an output. This architecture is commonly called a *multilayer perceptron*, often abbreviated as *MLP*. The neural network diagram for an MLP looks like Fig. 4.1.2.

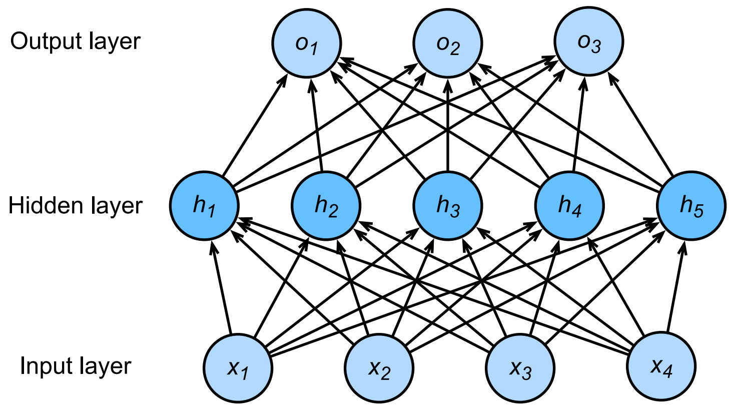


Fig. 4.1.2: Multilayer perceptron with hidden layers. This example contains a hidden layer with 5 hidden units in it.

The multilayer perceptron above has 4 inputs and 3 outputs, and the hidden layer in the middle contains 5 hidden units. Since the input layer does not involve any calculations, building this network would consist of implementing 2 layers of computation. The neurons in the input layer are fully connected to the inputs in the hidden layer. Likewise, the neurons in the hidden layer are fully connected to the neurons in the output layer.

From Linear to Nonlinear

We can write out the calculations that define this one-hidden-layer MLP in mathematical notation as follows:

$$\begin{aligned}\mathbf{h} &= \mathbf{W}_1\mathbf{x} + \mathbf{b}_1,\\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2,\\ \hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{o}).\end{aligned} \tag{4.1.2}$$

By adding another layer, we have added two new sets of parameters, but what have we gained in exchange? In the model defined above, we do not achieve anything for our troubles!

That is because our hidden units are just a linear function of the inputs and the outputs (pre-softmax) are just a linear function of the hidden units. A linear function of a linear function is itself a linear function. That means that for any values of the weights, we could just collapse out the hidden layer yielding an equivalent single-layer model using $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$ and $\mathbf{b} = \mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2$.

$$\mathbf{o} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W}\mathbf{x} + \mathbf{b}. \tag{4.1.3}$$

In order to get a benefit from multilayer architectures, we need another key ingredient—a nonlinearity σ to be applied to each of the hidden units after each layer's linear transformation. The most popular choice for the nonlinearity these days is the rectified linear unit (ReLU) $\max(x, 0)$. After incorporating these non-linearities it becomes impossible to merge layers.

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1),\\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2,\\ \hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{o}).\end{aligned} \tag{4.1.4}$$

Clearly, we could continue stacking such hidden layers, e.g., $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)$ and $\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1+\mathbf{b}_2)$ on top of each other to obtain a true multilayer perceptron.

Multilayer perceptrons can account for complex interactions in the inputs because the hidden neurons depend on the values of each of the inputs. It's easy to design a hidden node that does arbitrary computation, such as, for instance, logical operations on its inputs. Moreover, for certain choices of the activation function it's widely known that multilayer perceptrons are universal approximators. That means that even for a single-hidden-layer neural network, with enough nodes, and the right set of weights, we can model any function at all! *Actually learning that function is the hard part.*

Moreover, just because a single-layer network *can* learn any function does not mean that you should try to solve all of your problems with single-layer networks. It turns out that we can approximate many functions much more compactly if we use deeper (vs wider) neural networks. We'll get more into the math in a subsequent chapter, but for now let's actually build an MLP. In this example, we'll implement a multilayer perceptron with two hidden layers and one output layer.

Vectorization and Minibatch

As before, by the matrix $\mathbf{X}$, we denote a minibatch of inputs. The calculations to produce outputs from an MLP with two hidden layers can thus be expressed:

$$\begin{aligned} \mathbf{H}_1 &= \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1), \\ \mathbf{H}_2 &= \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2), \\ \mathbf{O} &= \text{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3). \end{aligned} \tag{4.1.5}$$

With some abuse of notation, we define the nonlinearity σ to apply to its inputs on a row-wise fashion, i.e., one observation at a time. Note that we are also using the notation for *softmax* in the same way to denote a row-wise operation. Often, as in this section, the activation functions that we apply to hidden layers are not merely row-wise, but component wise. That means that after computing the linear portion of the layer, we can calculate each nodes activation without looking at the values taken by the other hidden units. This is true for most activation functions (the batch normalization operation will be introduced in Section 7.5 is a notable exception to that rule).

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
npx.set_np()
```

4.1.2 Activation Functions

Because they are so fundamental to deep learning, before going further, let's take a brief look at some common activation functions.

ReLU Function

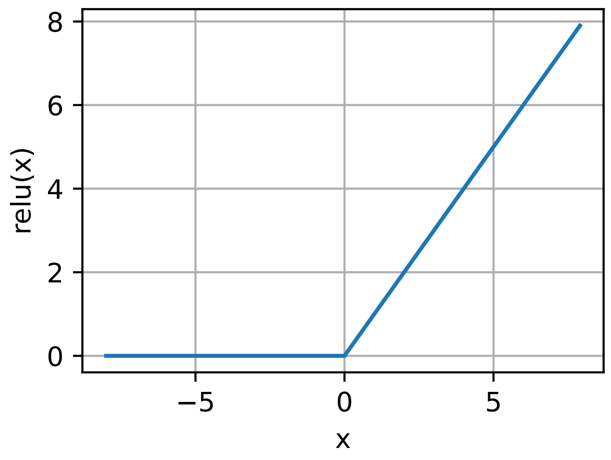
As stated above, the most popular choice, due to its simplicity of implementation and its efficacy in training is the rectified linear unit (ReLU). ReLUs provide a very simple nonlinear transformation. Given the element z, the function is defined as the maximum of that element and 0.

$$\text{ReLU}(z) = \max(z, 0). \tag{4.1.6}$$

It can be understood that the ReLU function retains only positive elements and discards negative elements (setting those nodes to 0). To get a better idea of what it looks like, we can plot it.

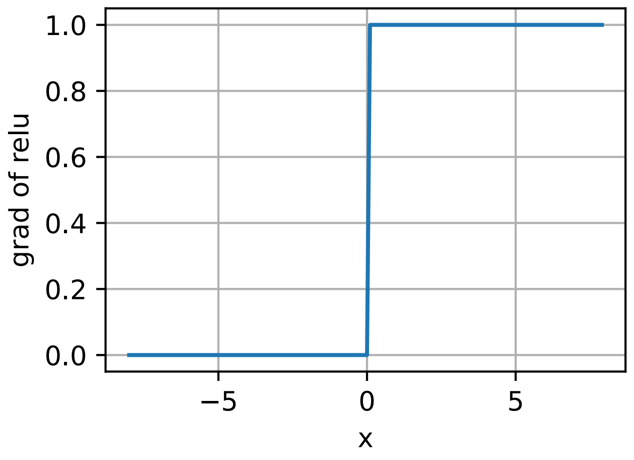
Because it is used so commonly, NDarray supports the `relu` function as a basic native operator. As you can see, the activation function is piece-wise linear.

```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.set_figsize((4, 2.5))
d2l.plot(x, y, 'x', 'relu(x)')
```



When the input is negative, the derivative of ReLU function is 0 and when the input is positive, the derivative of ReLU function is 1. Note that the ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we go with the left-hand-side (LHS) derivative and say that the derivative is 0 when the input is 0. We can get away with this because the input may never actually be zero. There is an old adage that if subtle boundary conditions matter, we are probably doing (*real*) mathematics, not engineering. That conventional wisdom may apply here. See the derivative of the ReLU function plotted below.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu')
```



Note that there are many variants to the ReLU function, such as the parameterized ReLU (pReLU) of He et al., 2015[63]. This variation adds a linear term to the ReLU, so some information still gets through, even when the argument is negative.

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{4.1.7}$$

The reason for using the ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it reduces the issue of the vanishing gradient problem (more on this later).

[63] https://arxiv.org/abs/1502.01852

Sigmoid Function

The sigmoid function transforms its inputs which take values in $\mathbb{R}$ to the interval $(0, 1)$. For that reason, the sigmoid is often called a *squashing* function: it squashes any input in the range (-inf, inf) to some value in the range (0, 1).

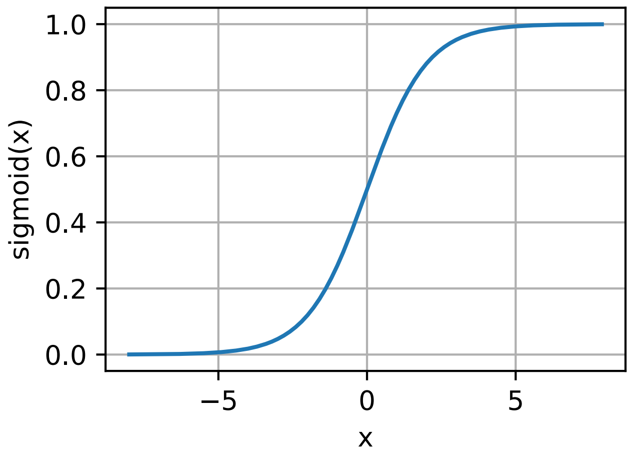
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{4.1.8}$$

In the earliest neural networks, scientists were interested in modeling biological neurons which either *fire* or *do not fire*. Thus the pioneers of this field, going all the way back to McCulloch and Pitts in the 1940s, were focused on thresholding units. A thresholding function takes either value 0 (if the input is below the threshold) or value 1 (if the input exceeds the threshold)

When attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit. Sigmoids are still common as activation functions on the output units, when we want to interpret the outputs as probabilities for binary classification problems (you can think of the sigmoid as a special case of the softmax) but the sigmoid has mostly been replaced by the simpler and easier to train ReLU for most use in hidden layers. In the "Recurrent Neural Network" chapter, we will describe how sigmoid units can be used to control the flow of information in a neural network thanks to its capacity to transform the value range between 0 and 1.

See the sigmoid function plotted below. When the input is close to 0, the sigmoid function approaches a linear transformation.

```python
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)')
```

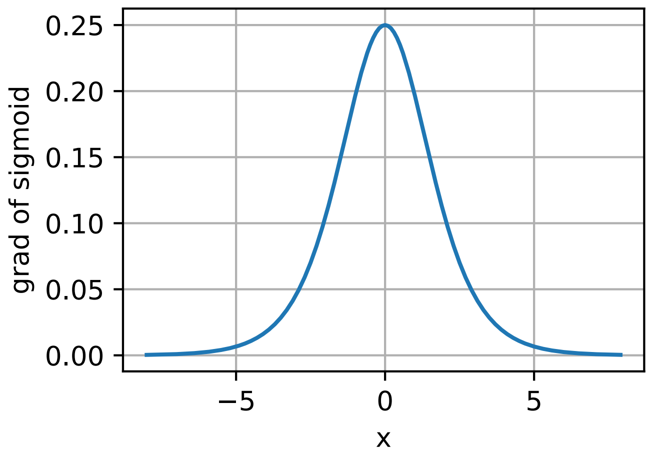


The derivative of sigmoid function is given by the following equation:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)\left(1 - \text{sigmoid}(x)\right). \tag{4.1.9}$$

The derivative of sigmoid function is plotted below. Note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25. As the input diverges from 0 in either direction, the derivative approaches 0.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of sigmoid')
```



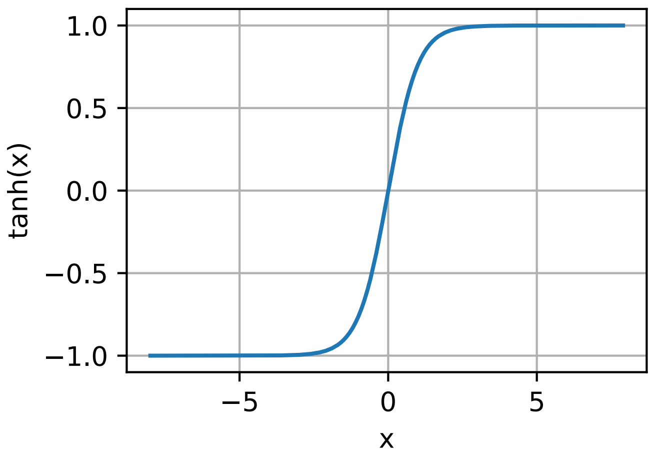
Tanh Function

Like the sigmoid function, the tanh (Hyperbolic Tangent) function also squashes its inputs, transforms them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{4.1.10}$$

We plot the tanh function blow. Note that as the input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system.

```
with autograd.record():
    y = np.tanh(x)
d2l.plot(x, y, 'x', 'tanh(x)')
```

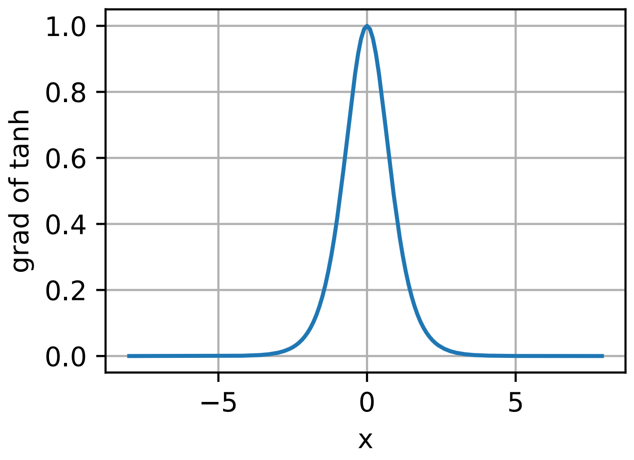


The derivative of the Tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{4.1.11}$$

The derivative of tanh function is plotted below. As the input nears 0, the derivative of the tanh function approaches a maximum of 1. And as we saw with the sigmoid function, as the input moves away from 0 in either direction, the derivative of the tanh function approaches 0.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of tanh')
```



In summary, we now know how to incorporate nonlinearities to build expressive multilayer neural network architectures. As a side note, your knowledge now already puts you in command of the state of the art in deep learning, circa 1990. In fact, you have an advantage over anyone working the 1990s, because you can leverage powerful open-source deep learning frameworks to build models rapidly, using only a few lines of code. Previously, getting these nets training required researchers to code up thousands of lines of C and Fortran.

Summary

- The multilayer perceptron adds one or multiple fully-connected hidden layers between the output and input layers and transforms the output of the hidden layer via an activation function.
- Commonly-used activation functions include the ReLU function, the sigmoid function, and the tanh function.

Exercises

1. Compute the derivative of the tanh and the pReLU activation function.
2. Show that a multilayer perceptron using only ReLU (or pReLU) constructs a continuous piecewise linear function.
3. Show that $\tanh(x) + 1 = 2\text{sigmoid}(2x)$.
4. Assume we have a multilayer perceptron *without* nonlinearities between the layers. In particular, assume that we have d input dimensions, d output dimensions and that one of the layers had only $d/2$ dimensions. Show that this network is less expressive (powerful) than a single layer perceptron.

5. Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems do you expect this to cause?

4.2 Implementation of Multilayer Perceptron from Scratch

Now that we know how multilayer perceptrons (MLPs) work in theory, let's implement them. First, we import the required packages.

```
import d2l
from mxnet import gluon, np, npx
npx.set_np()
```

To compare against the results we previously achieved with vanilla softmax regression, we continue to use the Fashion-MNIST image classification dataset.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

4.2.1 Initializing Model Parameters

Recall that this dataset contains 10 classes and that each image consists of a $28 \times 28 = 784$ grid of pixel values. Since we will be discarding the spatial structure (for now), we can just think of this as a classification dataset with 784 input features and 10 classes. In particular we will implement our MLP with one hidden layer and 256 hidden units. Note that we can regard both of these choices as *hyperparameters* that could be set based on performance on validation data. Typically, we will choose layer widths as powers of 2 to make everything align nicely in memory.

Again, we will represent our parameters with several `ndarrays`. Note that we now have one weight matrix and one bias vector *per layer*. As always, we must call `attach_grad` to allocate memory for the gradients with respect to these parameters.

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens))
b1 = np.zeros(num_hiddens)
W2 = np.random.normal(scale=0.01, size=(num_hiddens, num_outputs))
b2 = np.zeros(num_outputs)
params = [W1, b1, W2, b2]

for param in params:
    param.attach_grad()
```

4.2.2 Activation Function

To make sure we know how everything works, we will use the `maximum` function to implement ReLU ourselves, instead of invoking `npx.relu` directly.

```python
def relu(X):
    return np.maximum(X, 0)
```

4.2.3 The model

As in softmax regression, we will `reshape` each 2D image into a flat vector of length `num_inputs`. Finally, we can implement our model with just a few lines of code.

```python
def net(X):
    X = X.reshape(-1, num_inputs)
    H = relu(np.dot(X, W1) + b1)
    return np.dot(H, W2) + b2
```

4.2.4 The Loss Function

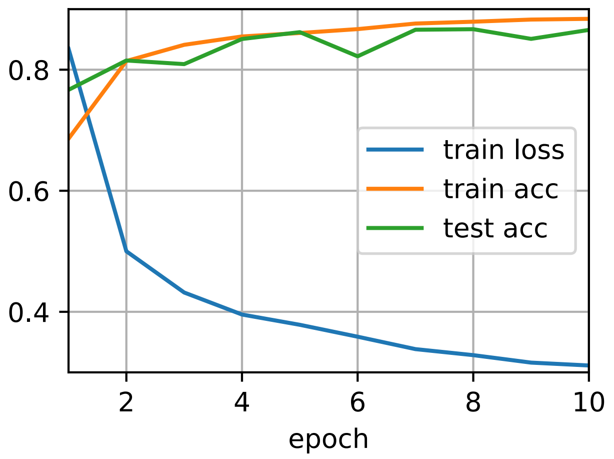
For better numerical stability and because we already know how to implement softmax regression completely from scratch in Section 3.6, we will use Gluon's integrated function for calculating the softmax and cross-entropy loss. Recall that we discussed some of these intricacies in Section 4.1. We encourage the interested reader to examing the source code for `mxnet.gluon.loss.SoftmaxCrossEntropyLoss` for more details.

```python
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

4.2.5 Training

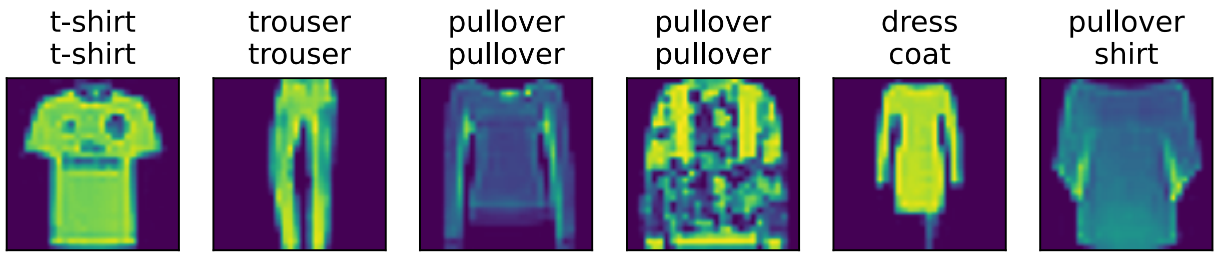
Steps for training the MLP are no different than for softmax regression. In the `d2l` package, we directly call the `train_ch3` function, whose implementation was introduced in Section 3.6. We set the number of epochs to 10 and the learning rate to 0.5.

```python
num_epochs, lr = 10, 0.5
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



To see how well we did, let's apply the model to some test data. If you are interested, compare the result to corresponding linear model in Section 3.6.

```
d2l.predict_ch3(net, test_iter)
```



This looks a bit better than our previous result, a good sign that we are on the right path.

Summary

We saw that implementing a simple MLP is easy, even when done manually. That said, with a large number of layers, this can get messy (e.g., naming and keeping track of the model parameters, etc).

Exercises

1. Change the value of the hyperparameter `num_hiddens` in order to see how this hyperparameter influences your results.
2. Try adding a new hidden layer to see how it affects the results.
3. How does changing the learning rate change the result?
4. What is the best result you can get by optimizing over all the parameters (learning rate, iterations, number of hidden layers, number of hidden units per layer)?

4.3 Concise Implementation of Multilayer Perceptron

Now that we learned how multilayer perceptrons (MLPs) work in theory, let's implement them. We begin, as always, by importing modules.

```
import d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

4.3.1 The Model

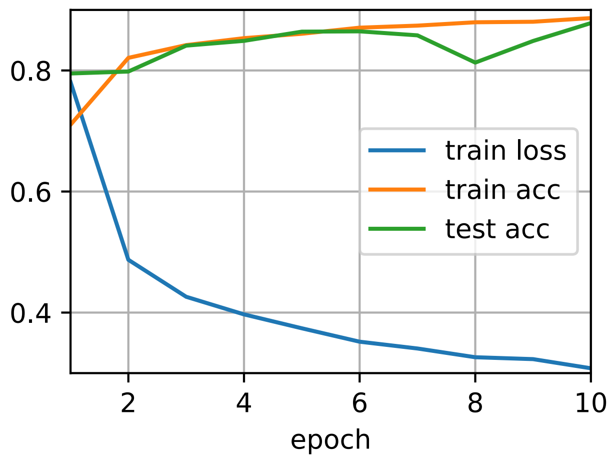
The only difference from our softmax regression implementation is that we add two `Dense` (fully-connected) layers instead of one. The first is our hidden layer, which has *256* hidden units and uses the ReLU activation function.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Again, note that as always, Gluon automatically infers the missing input dimensions to each layer.

Training the model follows the exact same steps as in our softmax regression implementation.

```
batch_size, num_epochs = 256, 10
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Exercises

1. Try adding a few more hidden layers to see how the result changes.
2. Try out different activation functions. Which ones work best?
3. Try out different initializations of the weights.

4.4 Model Selection, Underfitting and Overfitting

As machine learning scientists, our goal is to discover general patterns. Say, for example, that we wish to learn the pattern that associates genetic markers with the development of dementia in adulthood. It is easy enough to memorize our training set. Each person's genes uniquely identify them, not just among people represented in our dataset, but among all people on earth!

Given the genetic markers representing some person, we do not want our model to simply recognize "oh, that is Bob", and then output the classification, say among {*dementia*, *mild cognitive impairment*, *healthy*}, that corresponds to Bob. Rather, our goal is to discover patterns that capture regularities in the underlying population from which our training set was drawn. If we are successfully in this endeavor, then we could successfully assess risk even for individuals that we have never encountered before. This problem—how to discover patterns that *generalize*—is the fundamental problem of machine learning.

The danger is that when we train models, we access just a small sample of data. The largest public image datasets contain roughly one million images. And more often we have to learn from thousands or tens of thousands. In a large hospital system we might access hundreds of thousands of medical records. With finite samples, we always run the risk that we might discover *apparent* associations that turn out not to hold up when we collect more data.

Let's consider an extreme pathological case. Imagine that you want to learn to predict which people will repay their loans. A lender hires you as a data scientist to investigate, handing over the complete files on 100 applicants, 5 of which defaulted on their loans within 3 years. Realistically, the files might include hundreds of potential features, including income, occupation, credit score, length of employment etc. Moreover, say that they additionally hand over video footage of each applicant's interview with their lending agent.

Now suppose that after featurizing the data into an enormous design matrix, you discover that of the 5 applicants who default, all of them were wearing blue shirts during their interviews, while only 40% of general population wore blue shirts. There is a good chance that if you train a predictive model to predict default, it might rely upon blue-shirt-wearing as an important feature.

Even if in fact defaulters were no more likely to wear blue shirts than people in the general population, there's a $.4^5 = .01$ probability that we would observe all five defaulters wearing blue shirts. With just 5 positive examples of defaults and hundreds or thousands of features, we would probably find a large number of features that appear to be perfectly predictive of our labor just due to random chance. With an unlimited amount of data, we would expect these *spurious* associations to eventually disappear. But we seldom have that luxury.

The phenomena of fitting our training data more closely than we fit the underlying distribution is called overfitting, and the techniques used to combat overfitting are called regularization. In the previous sections, you might have observed this effect while experimenting with the Fashion-MNIST dataset. If you altered the model structure or the hyper-parameters during the experiment, you might have noticed that with enough nodes, layers, and training epochs, the model can eventually reach perfect accuracy on the training set, even as the accuracy on test data deteriorates.

4.4.1 Training Error and Generalization Error

In order to discuss this phenomenon more formally, we need to differentiate between *training error* and *generalization error*. The training error is the error of our model as calculated on the training dataset, while generalization error is the expectation of our model's error were we to apply it to an infinite stream of additional data points drawn from the same underlying data distribution as our original sample.

Problematically, *we can never calculate the generalization error exactly*. That is because the imaginary stream of infinite data is an imaginary object. In practice, we must *estimate* the generalization error by applying our model to an independent test set constituted of a random selection of data points that were withheld from our training set.

The following three thought experiments will help illustrate this situation better. Consider a college student trying to prepare for his final exam. A diligent student will strive to practice well and test her abilities using exams from previous years. Nonetheless, doing well on past exams is no guarantee that she will excel when it matters. For instance, the student might try to prepare by rote learning the answers to the exam questions. This requires the student to memorize many things. She might even remember the answers for past exams perfectly. Another student might prepare by trying to understand the reasons for giving certain answers. In most cases, the latter student will do much better.

Likewise, consider a model that simply uses a lookup table to answer questions. If the set of allowable inputs is discrete and reasonably small, then perhaps after viewing *many* training examples, this approach would perform well. Still this model has no ability to do better than random guessing when faced with examples that it has never seen before. In reality the input spaces are far too large to memorize the answers corresponding to every conceivable input. For example, consider the black and white 28×28 images. If each pixel can take one among 256 gray scale values, then there are 256^{784} possible images. That means that there are far more low-res grayscale thumbnail-sized images than there are atoms in the universe. Even if we could encounter this data, we could never afford to store the lookup table.

Last, consider the problem of trying to classify the outcomes of coin tosses (class 0: heads, class 1: tails) based on some contextual features that might be available. No matter what algorithm we come up with, because the generalization error will always be $\frac{1}{2}$. However, for most algorithms, we should expect our training error to be considerably lower, depending on the luck of the draw, even if we did not have any features! Consider the dataset {0, 1, 1, 1, 0, 1}. Our feature-less would have to fall back on always predicting the *majority class*, which appears from our limited sample to be *1*. In this case, the model that always predicts class 1 will incur an error of $\frac{1}{3}$, considerably better than our generalization error. As we increase the amount of data, the probability that the fraction of heads will deviate significantly from $\frac{1}{2}$ diminishes, and our training error would come to match the generalization error.

Statistical Learning Theory

Since generalization is the fundamental problem in machine learning, you might not be surprised to learn that many mathematicians and theorists have dedicated their lives to developing formal theories to describe this phenomenon. In their eponymous theorem[67], Glivenko and Cantelli derived the rate at which the training error converges to the generalization error. In a series of seminal papers, Vapnik and Chervonenkis[68] extended this theory to more general classes of functions. This work laid the foundations of Statistical Learning Theory[69].

In the *standard supervised learning setting*, which we have addressed up until now and will stick throughout most of this book, we assume that both the training data and the test data are drawn *independently* from *identical* distributions (commonly called the i.i.d. assumption). This means that the process that samples our data has no *memory*. The 2nd example drawn and the 3rd drawn are no more correlated than the 2nd and the 2-millionth sample drawn.

Being a good machine learning scientist requires thinking critically, and already you should be poking holes in this assumption, coming up with common cases where the assumption fails. What if we train a mortality risk predictor on data collected from patients at UCSF, and apply it on patients at Massachusetts General Hospital? These distributions are simply not identical. Moreover, draws might be correlated in time. What if we are classifying the topics of Tweets. The news cycle would create temporal dependencies in the topics being discussed violating any assumptions of independence.

Sometimes we can get away with minor violations of the i.i.d. assumption and our models will continue to work remarkably well. After all, nearly every real-world application involves at least some minor violation of the i.i.d. assumption, and yet we have useful tools for face recognition, speech recognition, language translation, etc.

Other violations are sure to cause trouble. Imagine, for example, if we tried to train a face recognition system by training it exclusively on university students and then want to deploy it as a tool for monitoring geriatrics in a nursing home population. This is unlikely to work well since college students tend to look considerably different from the elderly.

In subsequent chapters and volumes, we will discuss problems arising from violations of the i.i.d. assumption. For now, even taking the i.i.d. assumption for granted, understanding generalization is a formidable problem. Moreover, elucidating the precise theoretical foundations that might explain why deep neural networks generalize as well as they do continues to vexes the greatest minds in learning theory.

When we train our models, we attempt searching for a function that fits the training data as well as possible. If the function is so flexible that it can catch on to spurious patterns just as easily as to the true associations, then it might perform *too well* without producing a model that generalizes well to unseen data. This is precisely what we want to avoid (or at least control). Many of the techniques in deep learning are heuristics and tricks aimed at guarding against overfitting.

[67] https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem
[68] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory
[69] https://en.wikipedia.org/wiki/Statistical_learning_theory

Model Complexity

When we have simple models and abundant data, we expect the generalization error to resemble the training error. When we work with more complex models and fewer examples, we expect the training error to go down but the generalization gap to grow. What precisely constitutes model complexity is a complex matter. Many factors govern whether a model will generalize well. For example a model with more parameters might be considered more complex. A model whose parameters can take a wider range of values might be more complex. Often with neural networks, we think of a model that takes more training steps as more complex, and one subject to *early stopping* as less complex.

It can be difficult to compare the complexity among members of substantially different model classes (say a decision tree versus a neural network). For now, a simple rule of thumb is quite useful: A model that can readily explain arbitrary facts is what statisticians view as complex, whereas one that has only a limited expressive power but still manages to explain the data well is probably closer to the truth. In philosophy, this is closely related to Popper's criterion of falsifiability[70] of a scientific theory: a theory is good if it fits data and if there are specific tests which can be used to disprove it. This is important since all statistical estimation is post hoc[71], i.e., we estimate after we observe the facts, hence vulnerable to the associated fallacy. For now, we will put the philosophy aside and stick to more tangible issues.

In this section, to give you some intuition, we'll focus on a few factors that tend to influence the generalizability of a model class:

1. The number of tunable parameters. When the number of tunable parameters, sometimes called the *degrees of freedom*, is large, models tend to be more susceptible to overfitting.
2. The values taken by the parameters. When weights can take a wider range of values, models can be more susceptible to over fitting.
3. The number of training examples. It's trivially easy to overfit a dataset containing only one or two examples even if your model is simple. But overfitting a dataset with millions of examples requires an extremely flexible model.

4.4.2 Model Selection

In machine learning, we usually select our final model after evaluating several candidate models. This process is called model selection. Sometimes the models subject to comparison are fundamentally different in nature (say, decision trees vs linear models). At other times, we are comparing members of the same class of models that have been trained with different hyperparameter settings.

With multilayer perceptrons for example, we may wish to compare models with different numbers of hidden layers, different numbers of hidden units, and various choices of the activation functions applied to each hidden layer. In order to determine the best among our candidate models, we will typically employ a validation set.

[70] https://en.wikipedia.org/wiki/Falsifiability
[71] https://en.wikipedia.org/wiki/Post_hoc

Validation Dataset

In principle we should not touch our test set until after we have chosen all our hyper-parameters. Were we to use the test data in the model selection process, there is a risk that we might overfit the test data. Then we would be in serious trouble. If we overfit our training data, there is always the evaluation on test data to keep us honest. But if we overfit the test data, how would we ever know?

Thus, we should never rely on the test data for model selection. And yet we cannot rely solely on the training data for model selection either because we cannot estimate the generalization error on the very data that we use to train the model.

The common practice to address this problem is to split our data three ways, incorporating a *validation set* in addition to the training and test sets.

In practical applications, the picture gets muddier. While ideally we would only touch the test data once, to assess the very best model or to compare a small number of models to each other, real-world test data is seldom discarded after just one use. We can seldom afford a new test set for each round of experiments.

The result is a murky practice where the boundaries between validation and test data are worryingly ambiguous. Unless explicitly stated otherwise, in the experiments in this book we are really working with what should rightly be called training data and validation data, with no true test sets. Therefore, the accuracy reported in each experiment is really the validation accuracy and not a true test set accuracy. The good news is that we do not need too much data in the validation set. The uncertainty in our estimates can be shown to be of the order of $\mathcal{O}(n^{-\frac{1}{2}})$.

K-Fold Cross-Validation

When training data is scarce, we might not even be able to afford to hold out enough data to constitute a proper validation set. One popular solution to this problem is to employ *K-fold cross-validation*. Here, the original training data is split into K non-overlapping subsets. Then model training and validation are executed K times, each time training on $K-1$ subsets and validating on a different subset (the one not used for training in that round). Finally, the training and validation error rates are estimated by averaging over the results from the K experiments.

4.4.3 Underfitting or Overfitting?

When we compare the training and validation errors, we want to be mindful of two common situations: First, we want to watch out for cases when our training error and validation error are both substantial but there is a little gap between them. If the model is unable to reduce the training error, that could mean that our model is too simple (i.e., insufficiently expressive) to capture the pattern that we are trying to model. Moreover, since the *generalization gap* between our training and validation errors is small, we have reason to believe that we could get away with a more complex model. This phenomenon is known as underfitting.

On the other hand, as we discussed above, we want to watch out for the cases when our training error is significantly lower than our validation error, indicating severe overfitting. Note that overfitting is not always a bad thing. With deep learning especially, it is well known that the best predictive models often perform far better on training data than on holdout data. Ultimately, we usually care more about the validation error than about the gap between the training and validation errors.

Whether we overfit or underfit can depend both on the complexity of our model and the size of the available training datasets, two topics that we discuss below.

Model Complexity

To illustrate some classical intuition about overfitting and model complexity, we given an example using polynomials. Given training data consisting of a single feature x and a corresponding real-valued label y, we try to find the polynomial of degree d

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{4.4.1}$$

to estimate the labels y. This is just a linear regression problem where our features are given by the powers of x, the w_i given the model's weights, and the bias is given by w_0 since $x^0 = 1$ for all x. Since this is just a linear regression problem, we can use the squared error as our loss function.

A higher-order polynomial function is more complex than a lower order polynomial function, since the higher-order polynomial has more parameters and the model function's selection range is wider. Fixing the training dataset, higher-order polynomial functions should always achieve lower (at worst, equal) training error relative to lower degree polynomials. In fact, whenever the data points each have a distinct value of x, a polynomial function with degree equal to the number of data points can fit the training set perfectly. We visualize the relationship between polynomial degree and under- vs over-fitting in Fig. 4.4.1.

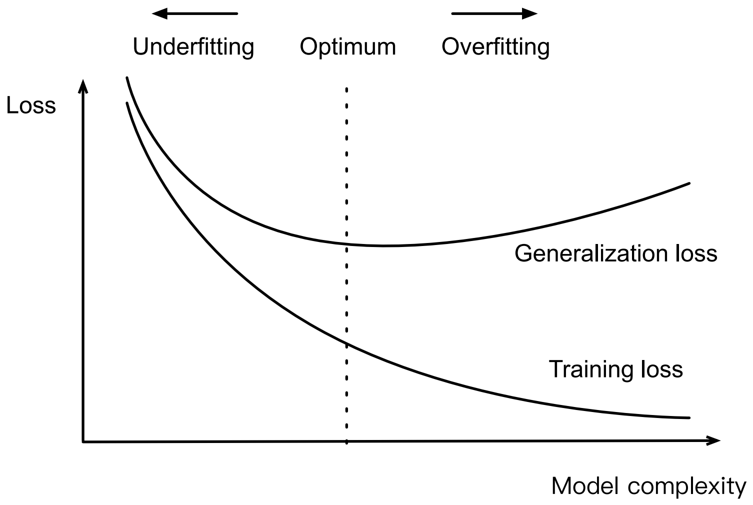


Fig. 4.4.1: Influence of Model Complexity on Underfitting and Overfitting

Dataset Size

The other big consideration to bear in mind is the dataset size. Fixing our model, the fewer samples we have in the training dataset, the more likely (and more severely) we are to encounter overfitting. As we increase the amount of training data, the generalization error typically decreases. Moreover, in general, more data never hurts. For a fixed task and data *distribution*, there is typically a relationship between model complexity and dataset size. Given more data, we might profitably attempt to fit a more complex model. Absent sufficient data, simpler models may be difficult to beat. For many tasks, deep learning only outperforms linear models when many thousands of training examples are available. In part, the current success of deep learning owes to the current abundance of massive datasets due to internet companies, cheap storage, connected devices, and the broad digitization of the economy.

4.4.4 Polynomial Regression

We can now explore these concepts interactively by fitting polynomials to data. To get started we will import our usual packages.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

Generating the Dataset

First we need data. Given x, we will use the following cubic polynomial to generate the labels on training and test data:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1). \tag{4.4.2}$$

The noise term ϵ obeys a normal distribution with a mean of 0 and a standard deviation of 0.1. We will synthesize 100 samples each for the training set and test set.

```
maxdegree = 20  # Maximum degree of the polynomial
n_train, n_test = 100, 100  # Training and test dataset sizes
true_w = np.zeros(maxdegree)  # Allocate lots of empty space
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

features = np.random.normal(size=(n_train + n_test, 1))
features = np.random.shuffle(features)
poly_features = np.power(features, np.arange(maxdegree).reshape(1, -1))
poly_features = poly_features / (
    npx.gamma(np.arange(maxdegree) + 1).reshape(1, -1))
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

For optimization, we typically want to avoid very large values of gradients, losses, etc. This is why the monomials stored in `poly_features` are rescaled from x^i to $\frac{1}{i!}x^i$. It allows us to avoid very large values for large exponents i. Factorials are implemented in Gluon using the Gamma function, where $n! = \Gamma(n+1)$.

Take a look at the first 2 samples from the generated dataset. The value 1 is technically a feature, namely the constant feature corresponding to the bias.

```
features[:2], poly_features[:2], labels[:2]
```

```
(array([[-0.03716067],
        [-1.1468065 ]]),
 array([[ 1.00000000e+00, -3.71606685e-02,  6.90457586e-04,
```

(continues on next page)

(continued from previous page)

```
        -8.55262169e-06,  7.94552903e-08, -5.90522298e-10,
         3.65736781e-12, -1.94157468e-14,  9.01877669e-17,
        -3.72381952e-19,  1.38379622e-21, -4.67479880e-24,
         1.44765544e-26, -4.13814215e-29,  1.09840096e-31,
        -2.72115417e-34,  6.31999553e-37, -1.38150092e-39,
         2.85164237e-42, -5.60519386e-45],
       [ 1.00000000e+00, -1.14680648e+00,  6.57582462e-01,
        -2.51373291e-01,  7.20691308e-02, -1.65298693e-02,
         3.15942708e-03, -5.17607376e-04,  7.41994299e-05,
        -9.45470947e-06,  1.08427218e-06, -1.13040933e-07,
         1.08030065e-08, -9.52996793e-10,  7.80644993e-11,
        -5.96832479e-12,  4.27782159e-13, -2.88578399e-14,
         1.83857564e-15, -1.10973155e-16]]),
 array([ 5.1432443 , -0.06415092]))
```

Training and Testing Model

Let's first implement a function to evaluate the loss on a given data.

```
# Saved in the d2l package for later use
def evaluate_loss(net, data_iter, loss):
    """Evaluate the loss of a model on the given dataset"""
    metric = d2l.Accumulator(2)  # sum_loss, num_examples
    for X, y in data_iter:
        metric.add(loss(net(X), y).sum(), y.size)
    return metric[0] / metric[1]
```

Now define the training function.

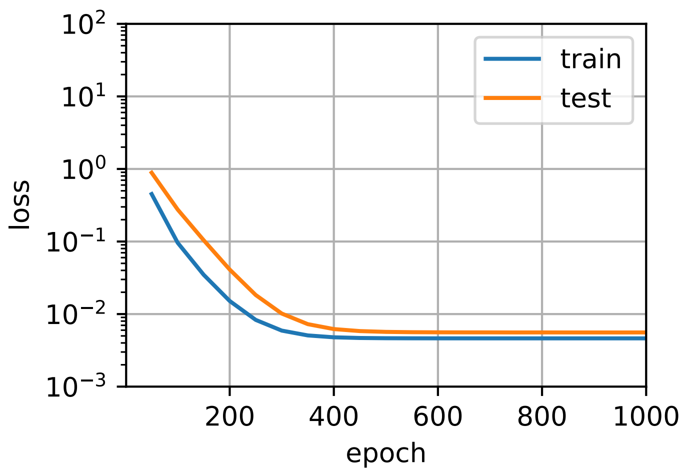
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=1000):
    loss = gluon.loss.L2Loss()
    net = nn.Sequential()
    # Switch off the bias since we already catered for it in the polynomial
    # features
    net.add(nn.Dense(1, use_bias=False))
    net.initialize()
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    test_iter = d2l.load_array((test_features, test_labels), batch_size,
                               is_train=False)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': 0.01})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(1, num_epochs+1):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch % 50 == 0:
            animator.add(epoch, (evaluate_loss(net, train_iter, loss),
                                 evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data().asnumpy())
```

Third-Order Polynomial Function Fitting (Normal)

We will begin by first using a third-order polynomial function with the same order as the data generation function. The results show that this model's training error rate when using the testing dataset is low. The trained model parameters are also close to the true values $w = [5, 1.2, -3.4, 5.6]$.

```
# Pick the first four dimensions, i.e., 1, x, x^2, x^3 from the polynomial
# features
train(poly_features[:n_train, 0:4], poly_features[n_train:, 0:4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.015855   1.1957889 -3.4174728  5.6198034]]
```

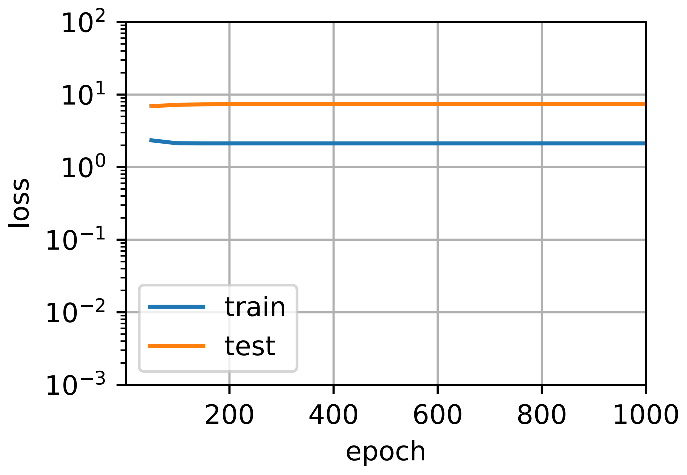


Linear Function Fitting (Underfitting)

Let's take another look at linear function fitting. After the decline in the early epoch, it becomes difficult to further decrease this model's training error rate. After the last epoch iteration has been completed, the training error rate is still high. When used to fit non-linear patterns (like the third-order polynomial function here) linear models are liable to underfit.

```
# Pick the first four dimensions, i.e., 1, x from the polynomial features
train(poly_features[:n_train, 0:3], poly_features[n_train:, 0:3],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.2635345  4.0288005 -3.9903946]]
```



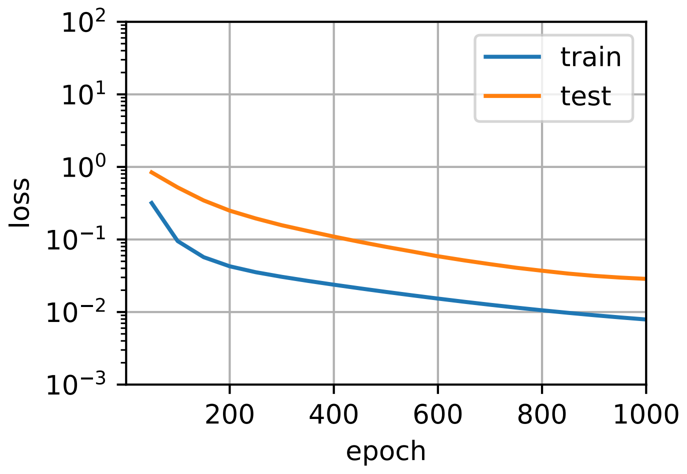
Insufficient Training (Overfitting)

Now let's try to train the model using a polynomial of too high degree. Here, there is insufficient data to learn that the higher-degree coefficients should have values close to zero. As a result, our overly-complex model is far too susceptible to being influenced by noise in the training data. Of course, our training error will now be low (even lower than if we had the right model!) but our test error will be high.

Try out different model complexities (n_degree) and training set sizes (n_subset) to gain some intuition of what is happening.

```
n_subset = 100  # Subset of data to train on
n_degree = 20  # Degree of polynomials
train(poly_features[1:n_subset, 0:n_degree],
      poly_features[n_train:, 0:n_degree], labels[1:n_subset],
      labels[n_train:])
```

```
weight: [[ 4.9478583   1.3325216  -3.2098305   5.042876   -0.4216669   1.347527
   0.07499089  0.19184932 -0.01922761  0.01773205 -0.05095618 -0.02382695
  -0.01497401 -0.04940015  0.06389714 -0.04761841 -0.04380195 -0.05188226
   0.05655775  0.01104914]]
```



In later chapters, we will continue to discuss overfitting problems and methods for dealing with them, such as weight decay and dropout.

Summary

- Since the generalization error rate cannot be estimated based on the training error rate, simply minimizing the training error rate will not necessarily mean a reduction in the generalization error rate. Machine learning models need to be careful to safeguard against overfitting such as to minimize the generalization error.
- A validation set can be used for model selection (provided that it is not used too liberally).
- Underfitting means that the model is not able to reduce the training error rate while overfitting is a result of the model training error rate being much lower than the testing dataset rate.
- We should choose an appropriately complex model and avoid using insufficient training samples.

Exercises

1. Can you solve the polynomial regression problem exactly? Hint: use linear algebra.
2. Model selection for polynomials
   - Plot the training error vs. model complexity (degree of the polynomial). What do you observe?
   - Plot the test error in this case.
   - Generate the same graph as a function of the amount of data?
3. What happens if you drop the normalization of the polynomial features x^i by $1/i!$. Can you fix this in some other way?
4. What degree of polynomial do you need to reduce the training error to 0?
5. Can you ever expect to see 0 generalization error?

4.5 Weight Decay

Now that we have characterized the problem of overfitting and motivated the need for capacity control, we can begin discussing some of the popular techniques used to these ends in practice. Recall that we can always mitigate overfitting by going out and collecting more training data, that can be costly and time consuming, typically making it impossible in the short run. For now, let's assume that we have already obtained as much high-quality data as our resources permit and focus on techniques aimed at limiting the capacity of the function classes under consideration.

In our toy example, we saw that we could control the complexity of a polynomial by adjusting its degree. However, most of machine learning does not consist of polynomial curve fitting. And moreover, even when we focus on polynomial regression, when we deal with high-dimensional data, manipulating model capacity by tweaking the degree d is problematic. To see why, note that for multivariate data we must generalize the concept of polynomials to include *monomials*, which are simply products of powers of variables. For example, $x_1^2 x_2$, and $x_3 x_5^2$ are both monomials of degree 3. The number of such terms with a given degree d blows up as a function of the degree d.

Concretely, for vectors of dimensionality D, the number of monomials of a given degree d is $\binom{D-1+d}{D-1}$. Hence, a small change in degree, even from say 1 to 2 or 2 to 3 would entail a massive blowup in the complexity of our model. Thus, tweaking the degree is too blunt a hammer. Instead, we need a more fine-grained tool for adjusting function complexity.

4.5.1 Squared Norm Regularization

Weight decay (commonly called *L2* regularization), might be the most widely-used technique for regularizing parametric machine learning models. The basic intuition behind weight decay is the notion that among all functions f, the function $f = 0$ is the simplest. Intuitively, we can then measure functions by their proximity to zero. But how precisely should we measure the distance between a function and zero? There is no single right answer. In fact, entire branches of mathematics, e.g., in functional analysis and the theory of Banach spaces are devoted to answering this issue.

For our present purposes, a very simple interpretation will suffice: We will consider a linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ to be simple if its weight vector is small. We can measure this via $||mathbf{f}w||^2$. One way of keeping the weight vector small is to add its norm as a penalty term to the problem of minimizing the loss. Thus we replace our original objective, *minimize the prediction error on the training labels*, with new objective, *minimize the sum of the prediction error and the penalty term*. Now, if the weight vector becomes too large, our learning algorithm will find more profit in minimizing the norm $||\mathbf{w}||^2$ versus minimizing the training error. That is exactly what we want. To illustrate things in code, let's revive our previous example from Section 3.1 for linear regression. There, our loss was given by

$$l(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{4.5.1}$$

Recall that $\mathbf{x}^{(i)}$ are the observations, $y^{(i)}$ are labels, and $(\mathbf{w}, b)$ are the weight and bias parameters respectively. To arrive at a new loss function that penalizes the size of the weight vector, we need to add $||mathbf{f}w||^2$, but how much should we add? To address this, we need to add a new hyperparameter, that we will call the *regularization constant* and denote by λ:

$$l(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{4.5.2}$$

This non-negative parameter $\lambda \geq 0$ governs the amount of regularization. For $\lambda = 0$, we recover our original loss function, whereas for $\lambda > 0$ we ensure that $\mathbf{w}$ cannot grow too large. The astute reader might wonder why we are squaring the norm of the weight vector. We do this for two reasons. First, we do it for computational convenience. By squaring the L2 norm, we remove the square root, leaving the sum of squares of each component of the weight vector. This is convenient because it is easy to compute derivatives of a sum of terms (the sum of derivatives equals the derivative of the sum).

Moreover, you might ask, why the L2 norm in the first place and not the L1 norm, or some other distance function. In fact, several other choices are valid and are popular throughout statistics.

While L2-regularized linear models constitute the classic *ridge regression* algorithm L1-regularized linear regression is a similarly fundamental model in statistics popularly known as *lasso regression*.

One mathematical reason for working with the L2 norm and not some other norm, is that it penalizes large components of the weight vector much more than it penalizes small ones. This encourages our learning algorithm to discover models which distribute their weight across a larger number of features, which might make them more robust in practice since they do not depend precariously on a single feature. The stochastic gradient descent updates for L2-regularized regression are as follows:

$$w \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right) \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right), \tag{4.5.3}$$

As before, we update $\mathbf{w}$ based on the amount by which our estimate differs from the observation. However, we also shrink the size of $\mathbf{w}$ towards 0. That is why the method is sometimes called “weight decay”: because the penalty term literally causes our optimization algorithm to *decay* the magnitude of the weight at each step of training. This is more convenient than having to pick the number of parameters as we did for polynomials. In particular, we now have a continuous mechanism for adjusting the complexity of f. Small values of λ correspond to unconstrained $\mathbf{w}$, whereas large values of λ constrain $\mathbf{w}$ considerably. Since we do not want to have large bias terms either, we often add b^2 as a penalty, too.

4.5.2 High-Dimensional Linear Regression

For high-dimensional regression it is difficult to pick the ‘right’ dimensions to omit. Weight-decay regularization is a much more convenient alternative. We will illustrate this below. First, we will generate some synthetic data as before

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01). \tag{4.5.4}$$

representing our label as a linear function of our inputs, corrupted by Gaussian noise with zero mean and variance 0.01. To observe the effects of overfitting more easily, we can make our problem high-dimensional, setting the data dimension to $d = 200$ and working with a relatively small number of training examples—here we will set the sample size to 20:

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

n_train, n_test, num_inputs, batch_size = 20, 100, 200, 1
true_w, true_b = np.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size, is_train=False)
```

4.5.3 Implementation from Scratch

Next, we will show how to implement weight decay from scratch. All we have to do here is to add the squared ℓ_2 penalty as an additional loss term added to the original target function. The squared norm penalty derives its name from the fact that we are adding the second power $\sum_i w_i^2$. The ℓ_2 is just one among an infinite class of norms call p-norms, many of which you might encounter in the future. In general, for some number p, the ℓ_p norm is defined as

$$\|\mathbf{w}\|_p^p := \sum_{i=1}^d |w_i|^p. \tag{4.5.5}$$

Initializing Model Parameters

First, we will define a function to randomly initialize our model parameters and run `attach_grad` on each to allocate memory for the gradients we will calculate.

```python
def init_params():
    w = np.random.normal(scale=1, size=(num_inputs, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    return [w, b]
```

Defining ℓ_2 Norm Penalty

Perhaps the most convenient way to implement this penalty is to square all terms in place and sum them up. We divide by 2 by convention (when we take the derivative of a quadratic function, the 2 and $1/2$ cancel out, ensuring that the expression for the update looks nice and simple).

```python
def l2_penalty(w):
    return (w**2).sum() / 2
```

Defining the Train and Test Functions

The following code defines how to train and test the model separately on the training dataset and the test dataset. Unlike the previous sections, here, the ℓ_2 norm penalty term is added when calculating the final loss function. The linear network and the squared loss have not changed since the previous chapter, so we will just import them via `d2l.linreg` and `d2l.squared_loss` to reduce clutter.

```python
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs + 1):
        for X, y in train_iter:
```

(continues on next page)

(continued from previous page)

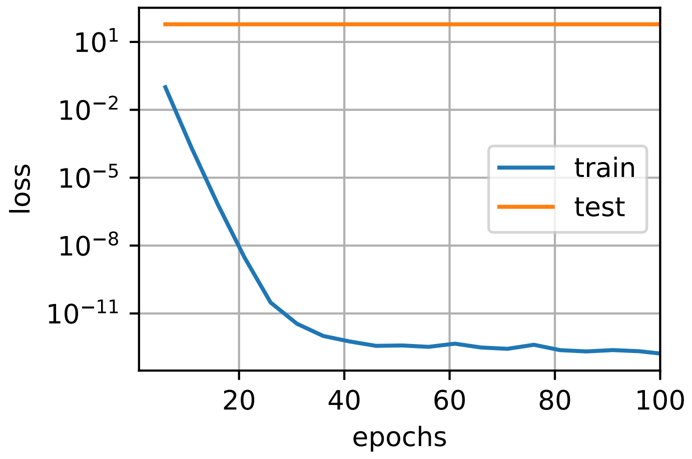
```
            with autograd.record():
                # The L2 norm penalty term has been added
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
        if epoch % 5 == 0:
            animator.add(epoch+1, (d2l.evaluate_loss(net, train_iter, loss),
                                   d2l.evaluate_loss(net, test_iter, loss)))
    print('l1 norm of w:', np.abs(w).sum())
```

Training without Regularization

Next, let's train and test the high-dimensional linear regression model. When `lambd = 0` we do not use weight decay. As a result, while the training error decreases, the test error does not. This is a perfect example of overfitting.

```
train(lambd=0)
```

```
l1 norm of w: 152.89598
```

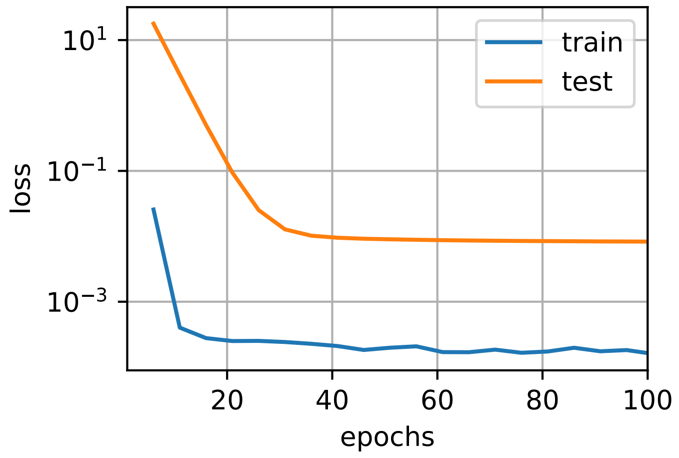


Using Weight Decay

The example below shows that even though the training error increased, the error on the test set decreased. This is precisely the improvement that we expect from using weight decay. While not perfect, overfitting has been mitigated to some extent. In addition, the ℓ_2 norm of the weight $\mathbf{w}$ is smaller than without using weight decay.

```
train(lambd=3)
```

```
l1 norm of w: 0.30536327
```



4.5.4 Concise Implementation

Because weight decay is ubiquitous in neural network optimization, Gluon makes it especially convenient, integrating weight decay into the optimization algorithm itself for easy use in combination with any loss function. Moreover, this integration serves a computational benefit, allowing implementation tricks to add weight decay to the algorithm, without any additional computational overhead. Since the weight decay portion of the update depends only on the current value of each parameter, and the optimizer must to touch each parameter once anyway.

In the following code, we specify the weight decay hyperparameter directly through the `wd` parameter when instantiating our `Trainer`. By default, Gluon decays both weights and biases simultaneously. Note that we can have *different* optimizers for different sets of parameters. For instance, we can have one `Trainer` with weight decay for the weights $\mathbf{w}$ and another without weight decay to take care of the bias b.

```python
def train_gluon(wd):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=1))
    loss = gluon.loss.L2Loss()
    num_epochs, lr = 100, 0.003
    # The weight parameter has been decayed. Weight names generally end with
    # "weight"
    trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                              {'learning_rate': lr, 'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
    trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                              {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs+1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            # Call the step function on each of the two Trainer instances to
            # update the weight and bias separately
            trainer_w.step(batch_size)
```

(continues on next page)

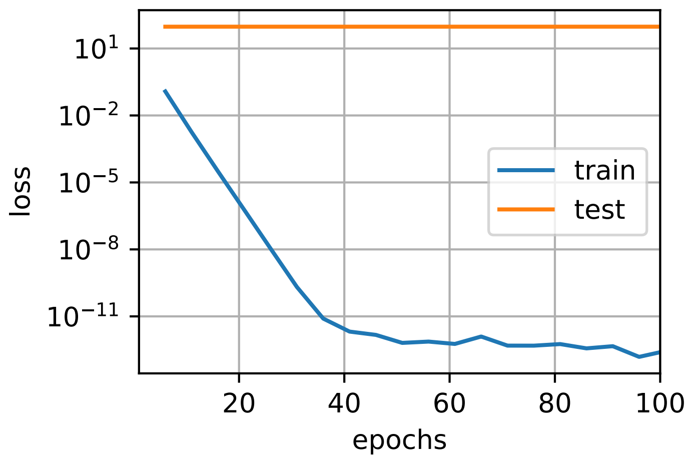
(continued from previous page)

```
            trainer_b.step(batch_size)
        if epoch % 5 == 0:
            animator.add(epoch+1, (d2l.evaluate_loss(net, train_iter, loss),
                                   d2l.evaluate_loss(net, test_iter, loss)))
    print('L1 norm of w:', np.abs(net[0].weight.data()).sum())
```

The plots look just the same as when we implemented weight decay from scratch but they run a bit faster and are easier to implement, a benefit that will become more pronounced for large problems.

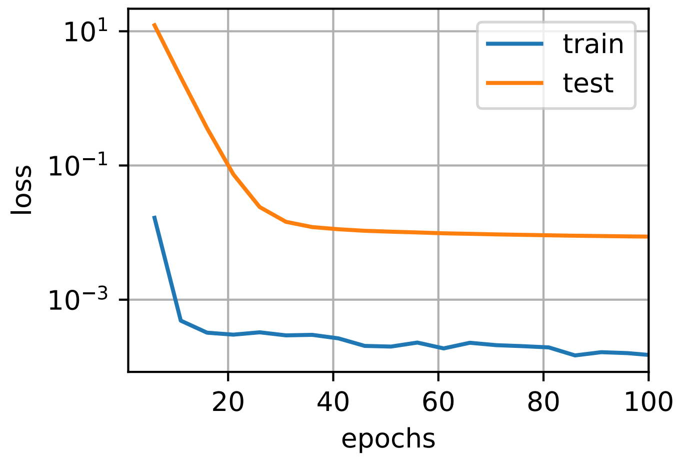
```
train_gluon(0)
```

```
L1 norm of w: 163.57817
```



```
train_gluon(3)
```

```
L1 norm of w: 0.3083761
```



So far, we only touched upon one notion of what constitutes a simple *linear* function. For nonlinear functions, what constitutes *simplicity* can be a far more complex question. For instance, there exist Reproducing Kernel Hilbert Spaces (RKHS)[73] which allow one to use many of the tools introduced for linear functions in a nonlinear context. Unfortunately, RKHS-based algorithms do not always scale well to massive amounts of data. For the purposes of this book, we limit ourselves to simply summing over the weights for different layers, e.g., via $\sum_l \|\mathbf{w}_l\|^2$, which is equivalent to weight decay applied to all layers.

Summary

- Regularization is a common method for dealing with overfitting. It adds a penalty term to the loss function on the training set to reduce the complexity of the learned model.
- One particular choice for keeping the model simple is weight decay using an ℓ_2 penalty. This leads to weight decay in the update steps of the learning algorithm.
- Gluon provides automatic weight decay functionality in the optimizer by setting the hyperparameter `wd`.
- You can have different optimizers within the same training loop, e.g., for different sets of parameters.

Exercises

1. Experiment with the value of λ in the estimation problem in this page. Plot training and test accuracy as a function of λ. What do you observe?
2. Use a validation set to find the optimal value of λ. Is it really the optimal value? Does this matter?
3. What would the update equations look like if instead of $\|\mathbf{w}\|^2$ we used $\sum_i |w_i|$ as our penalty of choice (this is called ℓ_1 regularization).
4. We know that $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Can you find a similar equation for matrices (mathematicians call this the Frobenius norm[74])?
5. Review the relationship between training error and generalization error. In addition to weight decay, increased training, and the use of a model of suitable complexity, what other ways can you think of to deal with overfitting?
6. In Bayesian statistics we use the product of prior and likelihood to arrive at a posterior via $P(w \mid x) \propto P(x \mid w)P(w)$. How can you identify $P(w)$ with regularization?

[73] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space
[74] https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm

4.6 Dropout

Just now, we introduced the classical approach of regularizing statistical models by penalizing the ℓ_2 norm of the weights. In probabilistic terms, we could justify this technique by arguing that we have assumed a prior belief that weights take values from a Gaussian distribution with mean 0. More intuitively, we might argue that we encouraged the model to spread out its weights among many features and rather than depending too much on a small number of potentially spurious associations.

4.6.1 Overfitting Revisited

Given many more features than examples, linear models can overfit. But when there are many more examples than features, we can generally count on linear models not to overfit. Unfortunately, the reliability with which linear models generalize comes at a cost: Linear models can't take into account interactions among features. For every feature, a linear model must assign either a positive or a negative weight. They lack the flexibility to account for context.

In more formal text, you'll see this fundamental tension between generalizability and flexibility discussed as the *bias-variance tradeoff*. Linear models have high bias (they can only represent a small class of functions), but low variance (they give similar results across different random samples of the data).

Deep neural networks take us to the opposite end of the bias-variance spectrum. Neural networks are so flexible because they aren't confined to looking at each feature individually. Instead, they can learn interactions among groups of features. For example, they might infer that "Nigeria" and "Western Union" appearing together in an email indicates spam but that "Nigeria" without "Western Union" does not.

Even when we only have a small number of features, deep neural networks are capable of overfitting. In 2017, a group of researchers presented a now well-known demonstration of the incredible flexibility of neural networks. They presented a neural network with randomly-labeled images (there was no true pattern linking the inputs to the outputs) and found that the neural network, optimized by SGD, could label every image in the training set perfectly.

Consider what this means. If the labels are assigned uniformly at random and there are 10 classes, then no classifier can get better than 10% accuracy on holdout data. Yet even in these situations, when there is no true pattern to be learned, neural networks can perfectly fit the training labels.

4.6.2 Robustness through Perturbations

Let's think briefly about what we expect from a good statistical model. We want it to do well on unseen test data. One way we can accomplish this is by asking what constitutes a "simple" model? Simplicity can come in the form of a small number of dimensions, which is what we did when discussing fitting a model with monomial basis functions. Simplicity can also come in the form of a small norm for the basis functions. This led us to weight decay (ℓ_2 regularization). Yet a third notion of simplicity that we can impose is that the function should be robust under small changes in the input. For instance, when we classify images, we would expect that adding some random noise to the pixels should be mostly harmless.

In 1995, Christopher Bishop formalized a form of this idea when he proved that training with input noise is equivalent to Tikhonov regularization (Bishop, 1995). In other words, he drew a clear mathematical connection between the requirement that a function be smooth (and thus simple), as we discussed in the section on weight decay, with and the requirement that it be resilient to perturbations in the input.

Then in 2014, Srivastava et al. (Srivastava et al., 2014) developed a clever idea for how to apply Bishop's idea to the *internal* layers of the network, too. Namely they proposed to inject noise into each layer of the network before calculating the subsequent layer during training. They realized that when training deep network with many layers, enforcing smoothness just on the input-output mapping misses out on what is happening internally in the network. Their proposed idea is called *dropout*, and it is now a standard technique that is widely used for training neural networks. Throughout training, on each iteration, dropout regularization consists simply of zeroing out some fraction (typically 50%) of the nodes in each layer before calculating the subsequent layer.

The key challenge then is how to inject this noise without introducing undue statistical *bias*. In other words, we want to perturb the inputs to each layer during training in such a way that the expected value of the layer is equal to the value it would have taken had we not introduced any noise at all.

In Bishop's case, when we are adding Gaussian noise to a linear model, this is simple: At each training iteration, just add noise sampled from a distribution with mean zero $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to the input $\mathbf{x}$, yielding a perturbed point $\mathbf{x}' = \mathbf{x} + \epsilon$. In expectation, $E[\mathbf{x}'] = \mathbf{x}$.

In the case of dropout regularization, one can debias each layer by normalizing by the fraction of nodes that were not dropped out. In other words, dropout with drop probability p is applied as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{4.6.1}$$

By design, the expectation remains unchanged, i.e., $E[h'] = h$. Intermediate activations h are replaced by a random variable h' with matching expectation. The name "dropout" arises from the notion that some neurons "drop out" of the computation for the purpose of computing the final result. During training, we replace intermediate activations with random variables.

4.6.3 Dropout in Practice

Recall the multilayer perceptron (Section 4.1) with a hidden layer and 5 hidden units. Its architecture is given by

$$\begin{aligned} h &= \sigma(W_1 x + b_1), \\ o &= W_2 h + b_2, \\ \hat{y} &= \text{softmax}(o). \end{aligned} \tag{4.6.2}$$

When we apply dropout to the hidden layer, we are essentially removing each hidden unit with probability p, (i.e., setting their output to 0). We can view the result as a network containing only a subset of the original neurons. In Fig. 4.6.1, h_2 and h_5 are removed. Consequently, the calculation of y no longer depends on h_2 and h_5 and their respective gradient also vanishes when performing backprop. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$. Intuitively, deep learning researchers often explain the intuition thusly: we do not want the network's output to depend too precariously on the exact activation pathway through the network. The original authors of the dropout technique described their intuition as an effort to prevent the *co-adaptation* of feature detectors.

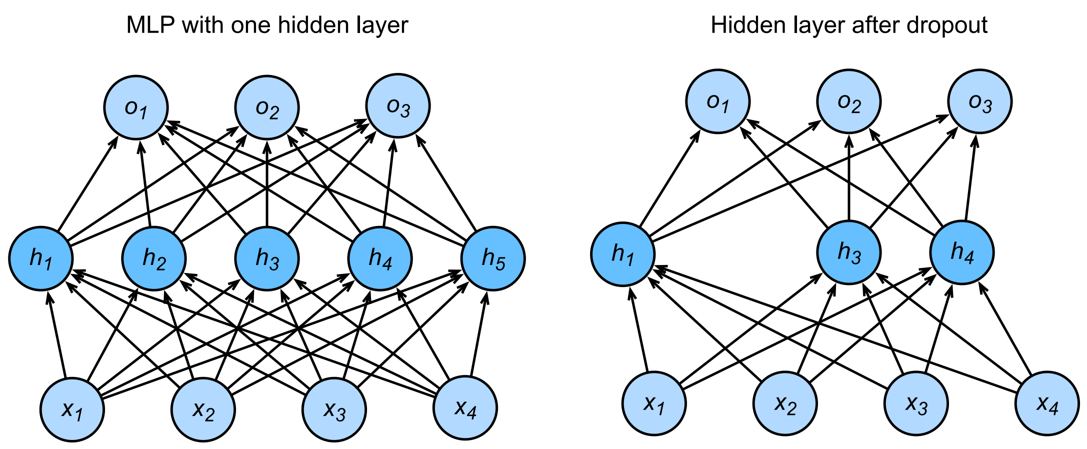


Fig. 4.6.1: MLP before and after dropout

At test time, we typically do not use dropout. However, we note that there are some exceptions: some researchers use dropout at test time as a heuristic approach for estimating the *confidence* of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident. For now we will put off the advanced topic of uncertainty estimation for subsequent chapters and volumes.

4.6.4 Implementation from Scratch

To implement the dropout function for a single layer, we must draw as many samples from a Bernoulli (binary) random variable as our layer has dimensions, where the random variable takes value 1 (keep) with probability $1 - p$ and 0 (drop) with probability p. One easy way to implement this is to first draw samples from the uniform distribution $U[0, 1]$. then we can keep those nodes for which the corresponding sample is greater than p, dropping the rest.

In the following code, we implement a `dropout` function that drops out the elements in the `ndarray` input `X` with probability `drop_prob`, rescaling the remainder as described above (dividing the survivors by `1.0-drop_prob`).

```
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

def dropout(X, drop_prob):
    assert 0 <= drop_prob <= 1
    # In this case, all elements are dropped out
    if drop_prob == 1:
        return np.zeros_like(X)
    mask = np.random.uniform(0, 1, X.shape) > drop_prob
    return mask.astype(np.float32) * X / (1.0-drop_prob)
```

We can test out the `dropout` function on a few examples. In the following lines of code, we pass our input `X` through the dropout operation, with probabilities 0, 0.5, and 1, respectively.

```
X = np.arange(16).reshape(2, 8)
print(dropout(X, 0))
```

(continues on next page)

(continued from previous page)

```
print(dropout(X, 0.5))
print(dropout(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
[[ 0.  0.  0.  0.  8. 10. 12.  0.]
 [16.  0. 20. 22.  0.  0.  0. 30.]]
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

Defining Model Parameters

Again, we can use the Fashion-MNIST dataset, introduced in Section 3.6. We will define a multilayer perceptron with two hidden layers. The two hidden layers both have 256 outputs.

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens1))
b1 = np.zeros(num_hiddens1)
W2 = np.random.normal(scale=0.01, size=(num_hiddens1, num_hiddens2))
b2 = np.zeros(num_hiddens2)
W3 = np.random.normal(scale=0.01, size=(num_hiddens2, num_outputs))
b3 = np.zeros(num_outputs)

params = [W1, b1, W2, b2, W3, b3]
for param in params:
    param.attach_grad()
```

Defining the Model

The model defined below concatenates the fully-connected layer and the activation function ReLU, using dropout for the output of each activation function. We can set the dropout probability of each layer separately. It is generally recommended to set a lower dropout probability closer to the input layer. Below we set it to 0.2 and 0.5 for the first and second hidden layer respectively. By using the `is_training` function described in Section 2.5, we can ensure that dropout is only active during training.

```
drop_prob1, drop_prob2 = 0.2, 0.5

def net(X):
    X = X.reshape(-1, num_inputs)
    H1 = npx.relu(np.dot(X, W1) + b1)
    # Use dropout only when training the model
    if autograd.is_training():
        # Add a dropout layer after the first fully connected layer
        H1 = dropout(H1, drop_prob1)
    H2 = npx.relu(np.dot(H1, W2) + b2)
    if autograd.is_training():
        # Add a dropout layer after the second fully connected layer
```

(continues on next page)

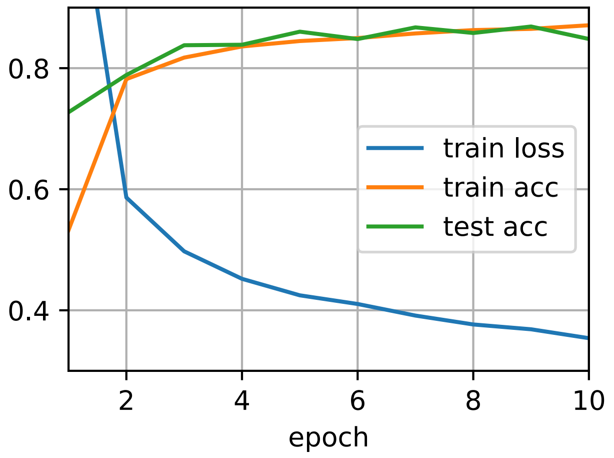
(continued from previous page)

```
        H2 = dropout(H2, drop_prob2)
    return np.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of multilayer perceptrons described previously.

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



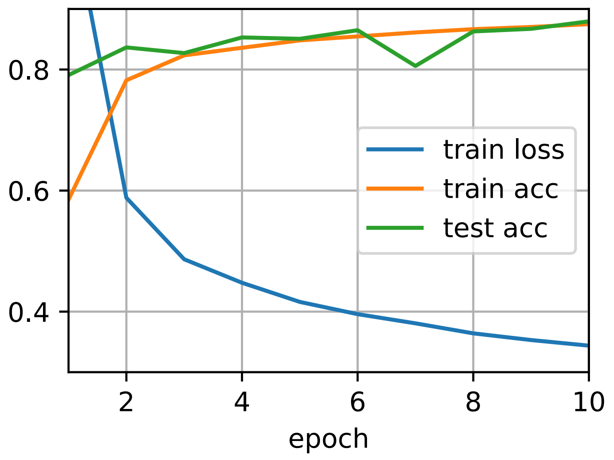
4.6.5 Concise Implementation

Using Gluon, all we need to do is add a `Dropout` layer (also in the `nn` package) after each fully-connected layer, passing in the dropout probability as the only argument to its constructor. During training, the `Dropout` layer will randomly drop out outputs of the previous layer (or equivalently, the inputs to the subsequent layer) according to the specified dropout probability. When MXNet is not in training mode, the `Dropout` layer simply passes the data through during testing.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation="relu"),
        # Add a dropout layer after the first fully connected layer
        nn.Dropout(drop_prob1),
        nn.Dense(256, activation="relu"),
        # Add a dropout layer after the second fully connected layer
        nn.Dropout(drop_prob2),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Next, we train and test the model.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often all three are used jointly.
- Dropout replaces an activation h with a random variable h' with expected value h and with variance given by the dropout probability p.
- Dropout is only used during training.

Exercises

1. Try out what happens if you change the dropout probabilities for layers 1 and 2. In particular, what happens if you switch the ones for both layers?
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. Compute the variance of the the activation random variables after applying dropout.
4. Why should you typically not using dropout?
5. If changes are made to the model to make it more complex, such as adding hidden layer units, will the effect of using dropout to cope with overfitting be more obvious?
6. Using the model in this section as an example, compare the effects of using dropout and weight decay. What if dropout and weight decay are used at the same time?
7. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?
8. Replace the dropout activation with a random variable that takes on values of $[0, \gamma/2, \gamma]$. Can you design something that works better than the binary dropout function? Why might you want to use it? Why not?

4.7 Forward Propagation, Backward Propagation, and Computational Graphs

In the previous sections, we used minibatch stochastic gradient descent to train our models. When we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. In other words, we implemented the calculations required for the model to generate output corresponding to come given input, but when it came time to calculate the gradients of each of our parameters, we invoked the `backward` function, relying on the `autograd` module to figure out what to do.

The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models would require recalculating lots of derivatives by hand. Even academic papers would too often have to allocate lots of page real estate to deriving update rules.

While we plan to continue relying on `autograd`, and we have already come a long way without every discussing how these gradients are calculated efficiently under the hood, it is important that you know how updates are actually calculated if you want to go beyond a shallow understanding of deep learning.

In this section, we will peel back the curtain on some of the details of backward propagation (more commonly called *backpropagation* or *backprop*). To convey some insight for both the techniques and how they are implemented, we will rely on both mathematics and computational graphs to describe the mechanics behind neural network computations. To start, we will focus our exposition on a simple multilayer perceptron with a single hidden layer and ℓ_2 norm regularization.

4.7.1 Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for the neural network within the models in the order from input layer to output layer. In the following, we work in detail through the example of a deep network with one hidden layer step by step. This is a bit tedious but it will serve us well when discussing what really goes on when we call `backward`.

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and there is no bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \qquad (4.7.1)$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After entering the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ into the activation function ϕ operated by the basic elements, we will obtain a hidden layer variable with the vector length of h,

$$\mathbf{h} = \phi(\mathbf{z}). \qquad (4.7.2)$$

The hidden variable $\mathbf{h}$ is also an intermediate variable. Assuming the parameters of the output layer only possess a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector length of q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{4.7.3}$$

Assuming the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y). \tag{4.7.4}$$

According to the definition of ℓ_2 norm regularization, given the hyperparameter λ, the regularization term is

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right), \tag{4.7.5}$$

where the Frobenius norm of the matrix is equivalent to the calculation of the L_2 norm after flattening the matrix to a vector. Finally, the model's regularized loss on a given data example is

$$J = L + s. \tag{4.7.6}$$

We refer to J as the objective function of a given data example and refer to it as the *objective function* in the following discussion.

4.7.2 Computational Graph of Forward Propagation

Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation. Fig. 4.7.1 contains the graph associated with the simple network described above. The lower-left corner signifies the input and the upper right corner the output. Notice that the direction of the arrows (which illustrate data flow) are primarily rightward and upward.

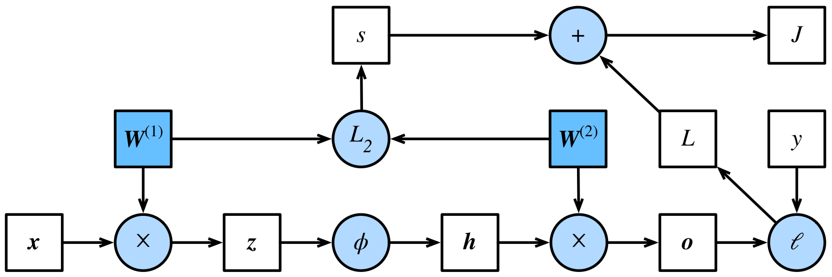


Fig. 4.7.1: Computational Graph

4.7.3 Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In general, back propagation calculates and stores the intermediate variables of an objective function related to each layer of the neural network and the gradient of the parameters in the order of the output layer to the input layer according to the 'chain rule' in calculus. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y}) = g \circ f(\mathsf{X})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z wrt. X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{4.7.7}$$

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions have been carried out. For vectors, this is straightforward: it is simply matrix-matrix multiplication and for higher dimensional tensors we use the appropriate counterpart. The operator prod hides all the notation overhead.

The parameters of the simple network with one hidden layer are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of backpropagation is to calculate the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To accomplish this, we will apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the compute graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ with respect to the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \tag{4.7.8}$$

Next, we compute the gradient of the objective function with respect to variable of the output layer **o** according to the chain rule.

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{4.7.9}$$

Next, we calculate the gradients of the regularization term with respect to both parameters.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{4.7.10}$$

Now we are able calculate the gradient $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{4.7.11}$$

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer's outputs $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ is given by

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{4.7.12}$$

Since the activation function ϕ applies elementwise, calculating the gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable **z** requires that we use the elementwise multiplication operator, which we denote by $\odot$.

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{4.7.13}$$

Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{4.7.14}$$

4.7.4 Training a Model

When training networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the compute graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why backpropagation requires significantly more memory than plain "inference"—we end up computing tensors as gradients and need to retain all the intermediate variables to invoke the chain rule. Another reason is that we typically train with minibatches containing more than one variable, thus more intermediate activations need to be stored.

Summary

- Forward propagation sequentially calculates and stores intermediate variables within the compute graph defined by the neural network. It proceeds from input to output layer.
- Back propagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.
- When training deep learning models, forward propagation and back propagation are interdependent.
- Training requires significantly more memory and storage.

Exercises

1. Assume that the inputs $\mathbf{x}$ are matrices. What is the dimensionality of the gradients?
2. Add a bias to the hidden layer of the model described in this section.
    - Draw the corresponding compute graph.
    - Derive the forward and backward propagation equations.
3. Compute the memory footprint for training and inference in model described in the current chapter.
4. Assume that you want to compute *second* derivatives. What happens to the compute graph? Is this a good idea?
5. Assume that the compute graph is too large for your GPU.
    - Can you partition it over more than one GPU?
    - What are the advantages and disadvantages over training on a smaller minibatch?

4.8 Numerical Stability and Initialization

In the past few sections, each model that we implemented required initializing our parameters according to some specified distribution. However, until now, we glossed over the details, taking the initialization hyperparameters for granted. You might even have gotten the impression that these choices are not especially important. However, the choice of initialization scheme plays a significant role in neural network learning, and can prove essentially to maintaining numerical stability. Moreover, these choices can be tied up in interesting ways with the choice of the activation function. Which nonlinear activation function we choose, and how we decide to initialize our parameters can play a crucial role in making the optimization algorithm converge rapidly. Failure to be mindful of these issues can lead to either exploding or vanishing gradients. In this section, we delve into these topics with greater detail and discuss some useful heuristics that you may use frequently throughout your career in deep learning.

4.8.1 Vanishing and Exploding Gradients

Consider a deep network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$. Each layer satisfies:

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}^t) \text{ and thus } \mathbf{o} = f_d \circ \ldots, \circ f_1(\mathbf{x}). \tag{4.8.1}$$

If all activations and inputs are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ associated with the function f_t at layer t simply as

$$\partial_{\mathbf{W}_t} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d}_{:= \mathbf{M}_d} \cdot \ldots, \cdot \underbrace{\partial_{\mathbf{h}^t} \mathbf{h}^{t+1}}_{:= \mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t} \mathbf{h}^t}_{:= \mathbf{v}_t}. \tag{4.8.2}$$

In other words, it is the product of $d - t$ matrices $\mathbf{M}_d \cdot \ldots, \cdot \mathbf{M}_t$ and the gradient vector $\mathbf{v}_t$. What happens is similar to the situation when we experienced numerical underflow when multiplying too many probabilities. At the time, we were able to mitigate the problem by switching from into log-space, i.e., by shifting the problem from the mantissa to the exponent of the numerical representation. Unfortunately the problem outlined in the equation above is much more serious: initially the matrices M_t may well have a wide variety of eigenvalues. They might be small, they might be large, and in particular, their product might well be *very large* or *very small*. This is not (only) a problem of numerical representation but it means that the optimization algorithm is bound to fail. It receives gradients that are either excessively large or excessively small. As a result the steps taken are either (i) excessively large (the *exploding* gradient problem), in which case the parameters blow up in magnitude rendering the model useless, or (ii) excessively small, (the *vanishing gradient problem*), in which case the parameters hardly move at all, and thus the learning process makes no progress.

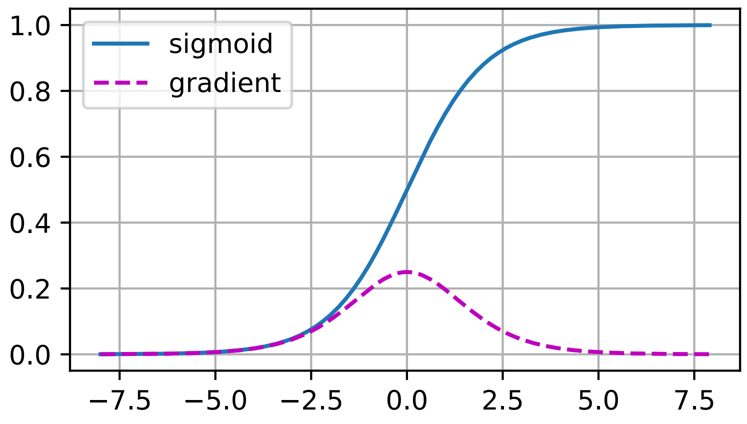
Vanishing Gradients

One major culprit in the vanishing gradient problem is the choices of the activation functions σ that are interleaved with the linear operations in each layer. Historically, the sigmoid function $(1 + \exp(-x))$ (introduced in Section 4.1) was a popular choice owing to its similarity to a thresholding function. Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that either fire or do not fire (biological neurons do not partially fire) seemed appealing. Let's take a closer look at the function to see why picking it might be problematic vis-a-vis vanishing gradients.

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
npx.set_np()

x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.sigmoid(x)
y.backward()

d2l.plot(x, [y, x.grad], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



As we can see, the gradient of the sigmoid vanishes both when its inputs are large and when they are small. Moreover, when we execute backward propagation, due to the chain rule, this means that unless we are in the Goldilocks zone, where the inputs to most of the sigmoids are in the range of, say $[-4, 4]$, the gradients of the overall product may vanish. When we have many layers, unless we are especially careful, we are likely to find that our gradient is cut off at *some* layer. Before ReLUs ($\max(0, x)$) were proposed as an alternative to squashing functions, this problem used to plague deep network training. As a consequence, ReLUs have become the default choice when designing activation functions in deep networks.

Exploding Gradients

The opposite problem, when gradients explode, can be similarly vexing. To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix. For the scale that we picked (the choice of the variance $\sigma^2 = 1$), the matrix product explodes. If this were to happen to us with a deep network, we would have no realistic chance of getting a gradient descent optimizer to converge.

```
M = np.random.normal(size=(4, 4))
print('A single matrix', M)
for i in range(100):
    M = np.dot(M, np.random.normal(size=(4, 4)))

print('After multiplying 100 matrices', M)
```

```
A single matrix [[ 2.2122064   1.1630787   0.7740038   0.4838046 ]
 [ 1.0434405   0.29956347  1.1839255   0.15302546]
 [ 1.8917114  -1.1688148  -1.2347414   1.5580711 ]
 [-1.771029   -0.5459446  -0.45138445 -2.3556297 ]]
After multiplying 100 matrices [[ 3.4459714e+23 -7.8040680e+23  5.9973287e+23  4.5229990e+23]
 [ 2.5275089e+23 -5.7240326e+23  4.3988473e+23  3.3174740e+23]
 [ 1.3731286e+24 -3.1097155e+24  2.3897773e+24  1.8022959e+24]
 [-4.4951040e+23  1.0180033e+24 -7.8232281e+23 -5.9000354e+23]]
```

Symmetry

Another problem in deep network design is the symmetry inherent in their parametrization. Assume that we have a deep network with one hidden layer with two units, say h_1 and h_2. In this case, we could permute the weights $\mathbf{W}_1$ of the first layer and likewise permute the weights of the output layer to obtain the same function. There is nothing special differentiating the first hidden unit vs the second hidden unit. In other words, we have permutation symmetry among the hidden units of each layer.

This is more than just a theoretical nuisance. Imagine what would happen if we initialized all of the parameters of some layer as $\mathbf{W}_l = c$ for some constant c. In this case, the gradients for all dimensions are identical: thus not only would each unit take the same value, but it would receive the same update. Stochastic gradient descent would never break the symmetry on its own and we might never be able to realize the networks expressive power. The hidden layer would behave as if it had only a single unit. As an aside, note that while SGD would not break this symmetry, dropout regularization would!

4.8.2 Parameter Initialization

One way of addressing, or at least mitigating the issues raised above is through careful initialization of the weight vectors. This way we can ensure that (at least initially) the gradients do not vanish and that they maintain a reasonable scale where the network weights do not diverge. Additional care during optimization and suitable regularization ensures that things never get too bad.

Default Initialization

In the previous sections, e.g., in Section 3.3, we used `net.initialize(init.Normal(sigma=0.01))` to initialize the values of our weights. If the initialization method is not specified, such as `net.initialize()`, MXNet will use the default random initialization method: each element of the weight parameter is randomly sampled with a uniform distribution $U[-0.07, 0.07]$ and the bias parameters are all set to 0. Both choices tend to work well in practice for moderate problem sizes.

Xavier Initialization

Let's look at the scale distribution of the activations of the hidden units h_i for some layer. They are given by

$$h_i = \sum_{j=1}^{n_\mathrm{in}} W_{ij} x_j. \tag{4.8.3}$$

The weights W_{ij} are all drawn independently from the same distribution. Furthermore, let's assume that this distribution has zero mean and variance σ^2 (this does not mean that the distribution has to be Gaussian, just that mean and variance need to exist). We do not really have much control over the inputs into the layer x_j but let's proceed with the somewhat unrealistic assumption that they also have zero mean and variance γ^2 and that they are independent of $\mathbf{W}$. In this case, we can compute mean and variance of h_i as follows:

$$\begin{aligned}
E[h_i] & = \sum_{j=1}^{n_\mathrm{in}} E[W_{ij} x_j] = 0, \\
E[h_i^2] & = \sum_{j=1}^{n_\mathrm{in}} E[W^2_{ij} x^2_j] \\
& = \sum_{j=1}^{n_\mathrm{in}} E[W^2_{ij}] E[x^2_j] \\
& = n_\mathrm{in} \sigma^2 \gamma^2.
\end{aligned} \tag{4.8.4}$$

One way to keep the variance fixed is to set $n_\mathrm{in} \sigma^2 = 1$. Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the top layers. That is, instead of $\mathbf{W}\mathbf{w}$, we need to deal with $\mathbf{W}^\top \mathbf{g}$, where $\mathbf{g}$ is the incoming gradient from the layer above. Using the same reasoning as for forward propagation, we see that the gradients' variance can blow up unless $n_\mathrm{out} \sigma^2 = 1$. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we simply try to satisfy:

$$\frac{1}{2} (n_\mathrm{in} + n_\mathrm{out}) \sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_\mathrm{in} + n_\mathrm{out}}}. \tag{4.8.5}$$

This is the reasoning underlying the eponymous Xavier initialization (Glorot & Bengio, 2010). It works well enough in practice. For Gaussian random variables, the Xavier initialization picks a normal distribution with zero mean and variance $\sigma^2 = 2/(n_\mathrm{in} + n_\mathrm{out})$. For uniformly distributed random variables $U[-a, a]$, note that their variance is given by $a^2/3$. Plugging $a^2/3$ into the condition on σ^2 yields that we should initialize uniformly with $U\left[-\sqrt{6/(n_\mathrm{in} + n_\mathrm{out})}, \sqrt{6/(n_\mathrm{in} + n_\mathrm{out})}\right]$.

Beyond

The reasoning above barely scratches the surface of modern approaches to parameter initialization. In fact, MXNet has an entire `mxnet.initializer` module implementing over a dozen different heuristics. Moreover, initialization continues to be a hot area of inquiry within research into the fundamental theory of neural network optimization. Some of these heuristics are especially suited for when parameters are tied (i.e., when parameters of in different parts the network are shared), for super-resolution, sequence models, and related problems. We recommend that the interested reader take a closer look at what is offered as part of this module, and investigate the recent research on parameter initialization. Perhaps you may come across a recent clever idea and contribute its implementation to MXNet, or you may even invent your own scheme!

Summary

- Vanishing and exploding gradients are common issues in very deep networks, unless great care is taking to ensure that gradients and parameters remain well controlled.
- Initialization heuristics are needed to ensure that at least the initial gradients are neither too large nor too small.
- The ReLU addresses one of the vanishing gradient problems, namely that gradients vanish for very large inputs. This can accelerate convergence significantly.
- Random initialization is key to ensure that symmetry is broken before optimization.

Exercises

1. Can you design other cases of symmetry breaking besides the permutation symmetry?
2. Can we initialize all weight parameters in linear regression or in softmax regression to the same value?
3. Look up analytic bounds on the eigenvalues of the product of two matrices. What does this tell you about ensuring that gradients are well conditioned?
4. If we know that some terms diverge, can we fix this after the fact? Look at the paper on LARS for inspiration (You et al., 2017).

4.9 Considering the Environment

So far, we have worked through a number of hands-on implementations fitting machine learning models to a variety of datasets. And yet, until now we skated over the matter of where are data comes from in the first place, and what we plan to ultimately *do* with the outputs from our models. Too often in the practice of machine learning, developers rush ahead with the development of models tossing these fundamental considerations aside.

Many failed machine learning deployments can be traced back to this situation. Sometimes the model does well as evaluated by test accuracy only to fail catastrophically in the real world when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst which perturbs the data distribution. Say for example that we trained a model to predict loan defaults, finding that the choice of footware was associated with risk of default (Oxfords indicate repayment, sneakers indicate default). We might be inclined to thereafter grant loans to all applicants wearing Oxfords and to deny all applicants wearing sneakers. But our ill-conceived leap from pattern recognition to decision-making and our failure to think critically about the environment might have disastrous consequences. For starters, as soon as we began making decisions based on footware, customers would catch on and change their behavior. Before long, all applicants would be wearing Oxfords, and yet there would be no coinciding improvement in credit-worthiness. Think about this deeply because similar issues abound in the application of machine learning: by introducing our model-based decisions to the environment, we might break the model.

In this section, we describe some common concerns and aim to get you started acquiring the critical thinking that you will need in order to detect these situations early, mitigate the 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 enter the realm of philosophy and grapple with difficult questions concerning ethics and informed consent.

4.9.1 Distribution Shift

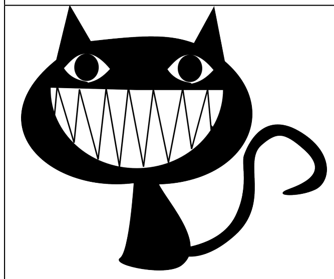
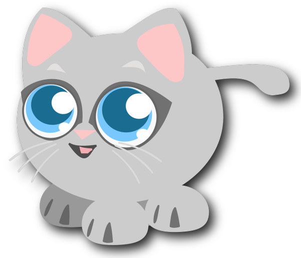
To begin, we return to the observational setting, putting aside for now the impacts of our actions on the environment. In the following sections, we take a deeper look at the various ways that data distributions might shift, and what might be done to salvage model performance. From the outset, we should warn that if the data-generating distribution $p(\mathbf{x}, y)$ can shift in arbitrary ways at any point in time, then learning a robust classifier is impossible. In the most pathological case, if the label definitions themselves can change at a moments notice: if suddenly what we called "cats" are now dogs and what we previously called "dogs" are now in fact cats, without any perceptible change in the distribution of inputs $p(\mathbf{x})$, then there is nothing we could do to detect the change or to correct our classifier at test time. Fortunately, under some restricted assumptions on the ways our data might change in the future, principled algorithms can detect shift and possibly even adapt, achieving higher accuracy than if we naively continued to rely on our original classifier.

Covariate Shift

One of the best-studied forms of distribution shift is *covariate shift*. Here we assume that although the distribution of inputs may change over time, the labeling function, i.e., the conditional distribution $P(y \mid \mathbf{x})$ does not change. While this problem is easy to understand its also easy to overlook it in practice. Consider the challenge of distinguishing cats and dogs. Our training data consists of images of the following kind:

cat	cat	dog	dog

At test time we are asked to classify the following images:

cat	cat	dog	dog
			

Obviously this is unlikely to work well. The training set consists of photos, while the test set contains only cartoons. The colors are not even realistic. Training on a dataset that looks substantially different from the test set without some plan for how to adapt to the new domain is a bad idea. Unfortunately, this is a very common pitfall. Statisticians call this *covariate shift* because the root of the problem owed to a shift in the distribution of features (i.e., of *covariates*). Mathematically, we could say that $P(\mathbf{x})$ changes but that $P(y \mid \mathbf{x})$ remains unchanged. Although its usefulness is not restricted to this setting, when we believe $\mathbf{x}$ causes y, covariate shift is usually the right assumption to be working with.

Label Shift

The converse problem emerges when we believe that what drives the shift is a change in the marginal distribution over the labels $P(y)$ but that the class-conditional distributions are invariant $P(\mathbf{x} \mid y)$. Label shift is a reasonable assumption to make when we believe that y causes $\mathbf{x}$. For example, commonly we want to predict a diagnosis given its manifestations. In this case we believe that the diagnosis causes the manifestations, i.e., diseases cause symptoms. Sometimes the label shift and covariate shift assumptions can hold simultaneously. For example, when the true labeling function is deterministic and unchanging, then covariate shift will always hold, including if label shift holds too. Interestingly, when we expect both label shift and covariate shift hold, it is often advantageous to work with the methods that flow from the label shift assumption. That is because these methods tend to involve manipulating objects that look like the label, which (in deep learning) tends to be comparatively easy compared to working with the objects that look like the input, which tends (in deep learning) to be a high-dimensional object.

Concept Shift

One more related problem arises in *concept shift*, the situation in which the very label definitions change. This sounds weird—after all, a *cat* is a *cat*. Indeed the definition of a cat might not change, but can we say the same about soft drinks? 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 definition of even this simple term as shown in Fig. 4.9.1.

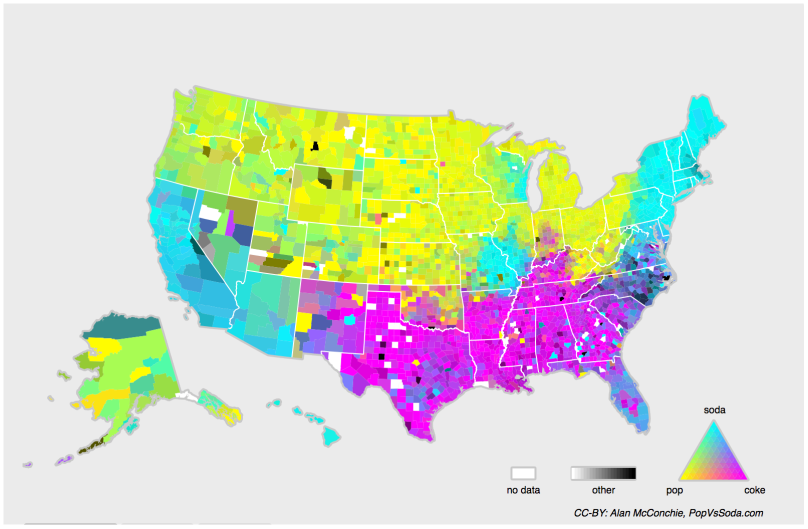


Fig. 4.9.1: Concept shift on soft drink names in the United States.

If we were to build a machine translation system, the distribution $P(y \mid x)$ might be different depending on our location. This problem can be tricky to spot. A saving grace is that often the $P(y \mid x)$ only shifts gradually.

Examples

Before we go into further detail and discuss remedies, we can discuss a number of situations where covariate and concept shift may not be so 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're ready for a successful career in medical diagnostics. Not so fast...

Many things could go wrong. In particular, the distributions that you work with for training and those that you encounter in the wild might differ considerably. This happened to an unfortunate startup, that Alex had the opportunity to consult for many years ago. They were developing a blood test for a disease that affects mainly older men and they'd managed to obtain a fair amount of blood samples from patients. It is considerably more difficult, though, to obtain blood samples from healthy men (mainly for ethical reasons). To compensate for that, they asked a large number of students on campus to donate blood and they performed their test. Then they asked me whether I could help them build a classifier to detect the disease. I told them that it would be very easy to distinguish between both datasets with near-perfect accuracy. After all, 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: Their sampling procedure made it likely that an extreme case of covariate shift would arise between the *source* and *target* distributions, and at that, one that could not be corrected by conventional means. In other words, training and test data were so different that nothing useful could be done and they had wasted significant amounts of money.

Self Driving Cars

Say a company wanted to build a machine learning system for self-driving cars. One of the key components 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 so-trained classifier worked "perfectly". Unfortunately, all it had learned was to distinguish trees with shadows from trees without shadows—the first set of pictures was taken in the early morning, the second one at noon.

Nonstationary distributions

A much more subtle situation arises when the distribution changes slowly and the model is not updated adequately. Here 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 wisen 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 it keeps on recommending 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 USA 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.

In short, 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.

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 x_i are drawn from some *target* distribution $q(\mathbf{x})$ rather than the *source* distribution $p(\mathbf{x})$. To make progress, we need to reflect about what exactly is happening during training: we iterate over training data and associated labels $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and update the weight vectors of the model after every minibatch. We sometimes additionally apply some penalty to the parameters, using weight decay, dropout, or some other related technique. This means that we largely minimize the loss on the training.

$$\underset{w}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(x_i, y_i, f(x_i)) + \text{some penalty}(w). \tag{4.9.1}$$

Statisticians call the first term an *empirical average*, i.e., an average computed over the data drawn from $P(x)P(y \mid x)$. If the data is drawn from the "wrong" distribution q, we can correct for that by using the following simple identity:

$$\int p(\mathbf{x}) f(\mathbf{x}) dx = \int q(\mathbf{x}) f(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} dx. \tag{4.9.2}$$

In other words, we need to re-weight each instance by the ratio of probabilities that it would have been drawn from the correct distribution $\beta(\mathbf{x}) := p(\mathbf{x})/q(\mathbf{x})$. 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 training data, and the one used for generating the training set q (the latter is trivially available). Note however, that we only need samples $\mathbf{x} \sim q(\mathbf{x})$; we do not to access labels $y \sim q(y)$.

In this case, there exists a very effective approach that will give almost as good results: logistic regression. This is all that is needed to compute estimate probability ratios. We learn a classifier to distinguish between data drawn from $p(\mathbf{x})$ and data drawn from $q(x)$. 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 the 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, denoted by $\mathbf{x}_i \sim p(\mathbf{x})$ and $\mathbf{x}'_i \sim q(\mathbf{x})$ respectively. Now denote by z_i labels which are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{4.9.3}$$

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

$$\beta(\mathbf{x}) = \frac{1/(1 + \exp(-f(\mathbf{x})))}{\exp(-f(\mathbf{x})/(1 + \exp(-f(\mathbf{x})))} = \exp(f(\mathbf{x})). \tag{4.9.4}$$

As a result, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a reweighted minimization problem where we weigh terms by β, e.g., via the head gradients. Here's a prototypical algorithm for that purpose which uses an unlabeled training set X and test set Z:

1. Generate training set with $\{(\mathbf{x}_i, -1)...(\mathbf{z}_j, 1)\}$.
2. Train binary classifier using logistic regression to get function f.
3. Weigh training data using $\beta_i = \exp(f(\mathbf{x}_i))$ or better $\beta_i = \min(\exp(f(\mathbf{x}_i)), c)$.
4. Use weights β_i for training on X with labels Y.

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

Generative Adversarial Networks use a very similar idea to that described above to engineer a *data generator* that outputs data that cannot be distinguished from examples sampled from a reference dataset. In these approaches, we use one network, f to distinguish real versus fake data and a second network g that tries to fool the discriminator f into accepting fake data as real. We will discuss this in much more detail later.

Label Shift Correction

For the discussion of label shift, we will assume for now that we are dealing with a k-way multiclass classification task. When the distribution of labels shifts over time $p(y) \neq q(y)$ but the class-conditional distributions stay the same $p(\mathbf{x}) = q(\mathbf{x})$, our importance weights will correspond to the label likelihood ratios $q(y)/p(y)$. 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 are often high-dimensional perceptual objects like images, while the labels are often easier to work, say vectors whose length corresponds to the number of classes).

To estimate calculate 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 C, is simply a $k \times k$ matrix where each column corresponds to the *actual* label and each row corresponds to our model's predicted label. Each cell's value c_{ij} is the fraction of predictions where the true label was j *and* our model predicted y.

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 models predictions at test time together, yielding the mean model output μ_y.

It turns out that under some mild conditions— if our classifier was reasonably accurate in the first place, if the target data contains only classes of images that we have seen before, and if the label shift assumption holds in the first place (far the strongest assumption here), then we can recover the test set label distribution by solving a simple linear system $C \cdot q(y) = \mu_y$. If our classifier is sufficiently accurate to begin with, then the confusion C will be invertible, and we get a solution $q(y) = C^{-1}\mu_y$. Here we abuse notation a bit, using $q(y)$ to denote the vector of label frequencies. Because we observe the labels on the source data, it is easy to estimate the distribution $p(y)$. Then for any training example i with label y, we can take the ratio of our estimates $\hat{q}(y)/\hat{p}(y)$ to calculate the weight w_i, and plug this into the weighted risk minimization algorithm above.

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 cameras 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.

4.9.2 A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in $p(x)$ and in $P(y \mid x)$, we can now consider some other aspects of machine learning problems formulation.

- **Batch Learning.** Here we have access to training data and labels $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, which we use to train a network $f(x, w)$. Later on, we deploy this network to score new data (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 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 (x_i, y_i) arrives one sample at a time. More specifically, assume that we first observe x_i, then we need to come up with an estimate $f(x_i, w)$ and only once we have done this, we observe y_i and with it, we receive a reward (or incur a loss), given our decision. Many real problems fall into this category. E.g. we need to predict tomorrow's stock price, this allows us to trade based on that estimate and at the end of the day we find out whether our estimate allowed us to make a profit. In other words, we have the following cycle where we are continuously improving our model given new observations.

$$\text{model } f_t \longrightarrow \text{data } x_t \longrightarrow \text{estimate } f_t(x_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(x_t)) \longrightarrow \text{model } f_{t+1} \tag{4.9.5}$$

- **Bandits.** They 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 (and nonadversarial Reinforcement Learning).** In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it'll just remember and the response will depend on what happened before. E.g. 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 him previously (e.g., he will read most news only once). Many such algorithms form a model of the environment in which they act such as to make their decisions appear less random (i.e., to reduce variance).
- **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. Likewise, we might want to build a good controller for autonomous cars. The 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, trying to cooperate with it, etc.

One key distinction between the different situations above is that the same strategy that might have worked throughout in the case of a stationary environment, might not work throughout when the environment can adapt. For instance, an arbitrage opportunity discovered by a trader is likely to disappear once he starts exploiting it. 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 to deal with concept shift, i.e., when the problem that he is trying to solve changes over time.

4.9.3 Fairness, Accountability, and Transparency in Machine Learning

Finally, it is important to remember that when you deploy machine learning systems you are not simply minimizing negative log likelihood or maximizing accuracy—you are automating some kind of decision process. Often the automated decision-making systems that we deploy can have consequences for those subject to its decisions. If we are deploying a medical diagnostic system, we need to know for which populations it may work and which it may not. Overlooking foreseeable risks to the welfare of a subpopulation would run afoul of basic ethical principles. Moreover, "accuracy" is seldom the right metric. When translating predictions in to actions we will often want to take into account the potential cost sensitivity of erring in various ways. If one way that you might classify an image could be perceived as a racial sleight, while misclassification to a different category would be harmless, then you might want to adjust your 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, if prediction systems are applied naively to predictive policing, allocating patrol officers accordingly, a vicious cycle might emerge. Neighborhoods that have more crimes, get more patrols, get more crimes discovered, get more training data, get yet more confident predictions, leading to even more patrols, even more crimes discovered, etc. 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 what news someone is exposed to be determined by which

Facebook pages they have *Liked*? These are just a few among the many profound ethical dilemmas that you might encounter in a career in machine learning.

Summary

- In many cases training and test set do not come from the same distribution. This is called covariate shift.
- Covariate shift can be detected and corrected if the shift is not too severe. Failure to do so leads to nasty surprises at test time.
- In some cases the environment *remembers* what we did and will respond in unexpected ways. We need to account for that when building models.

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. What could go wrong if training and test set are very different? What would happen to the sample weights?

4.10 Predicting House Prices on Kaggle

In the previous sections, we introduced the basic tools for building deep networks and performing capacity control via dimensionality-reduction, weight decay and dropout. You are now ready to put all this knowledge into practice by participating in a Kaggle competition. Predicting house prices[80] is a great place to start: the data is reasonably generic and does not have the kind of rigid structure that might require specialized models the way images or audio might. This dataset, collected by Bart de Cock in 2011 (DeCock, 2011), is considerably larger than the famous the Boston housing dataset[81] of Harrison and Rubinfeld (1978). It boasts both more examples and more features, covering house prices in Ames, IA from the period of 2006-2010.

In this section, we will walk you through details of data preprocessing, model design, hyperparameter selection and tuning. We hope that through a hands-on approach, you will be able to observe the effects of capacity control, feature extraction, etc. in practice. This experience is vital to gaining intuition as a data scientist.

[80] https://www.kaggle.com/c/house-prices-advanced-regression-techniques
[81] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

4.10.1 Obtaining Data and Caching

Throughout the book we will train models on various downloaded datasets. Here we developed several utility functions to facilitate data downloading. First, we will maintain a dictionary `DATA_HUB` that maps a string name to a URL with the SHA-1[82] of the content in the URL. If the original datasets are not convenient to download, we may host them on the `DATA_URL` site.

```python
import os
from mxnet import gluon
import zipfile
import tarfile

# Saved in the d2l package for later use
DATA_HUB = dict()
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'
```

The `download` method downloads the URL associated with the string name stored in `DATA_HUB` into a cache directory, which is `../data` in default. If the file already exists in the directory and its `SHA-1` matches the one specified in `DATA_HUB`, then the cached file is used without downloading. This function then returns the filename of the downloaded file.

```python
# Saved in the d2l package for later use
def download(name, cache_dir='../data'):
    """Download a file inserted into DATA_HUB, return the local filename"""
    assert name in DATA_HUB, "%s doesn't exist" % name
    url, sha1 = DATA_HUB[name]
    if not os.path.exists(cache_dir):
        os.makedirs(cache_dir)
    return gluon.utils.download(url, cache_dir, sha1_hash=sha1)
```

We also implement two additional methods, one to download and extract a `tar` or a `zip` file. The other one downloads all files in the `DATA_HUB`, namely almost all datasets needed in this book, into the cache directory. It's useful when your network is slow.

```python
# Saved in the d2l package for later use
def download_extract(name, folder=None):
    """Download and extract a .zip or a .tar file"""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext == '.tar' or ext == '.gz':
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip and tar files can be extracted'
    fp.extractall(base_dir)
    if folder:
        return base_dir + '/' + folder + '/'
    else:
        return data_dir + '/'
```

(continues on next page)

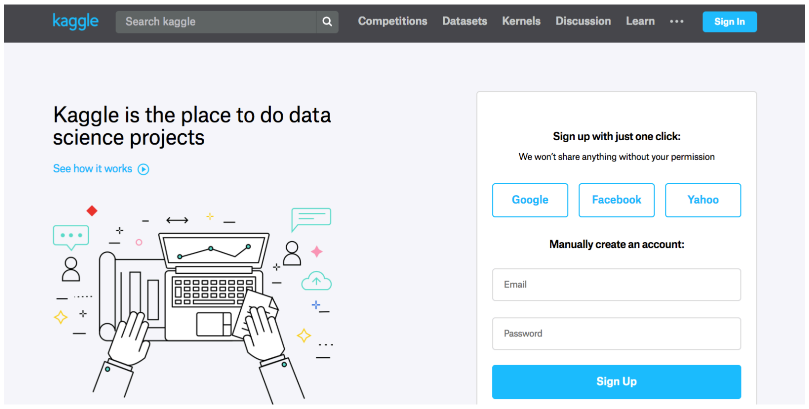
[82] https://en.wikipedia.org/wiki/SHA-1

(continued from previous page)

```
# Saved in the d2l package for later use
def download_all():
    """Download all files in the DATA_HUB"""
    for name in DATA_HUB:
        download(name)
```

4.10.2 Kaggle

Kaggle[83] is a popular platform for machine learning competitions. It combines data, code and users in a way to allow for both collaboration and competition. While leaderboard chasing can sometimes get out of control, there is also a lot to be said for the objectivity in a platform that provides fair and direct quantitative comparisons between your approaches and those devised by your competitors. Moreover, you can checkout the code from (some) other competitors' submissions and pick apart their methods to learn new techniques. If you want to participate in one of the competitions, you need to register for an account as shown in Fig. 4.10.1 (do this now!).



Kaggle is the place to do data science projects

See how it works

Sign up with just one click:

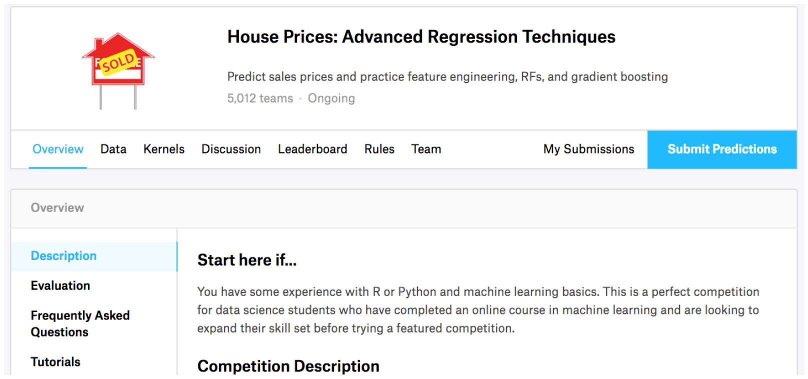
We won't share anything without your permission

Manually create an account:

Fig. 4.10.1: Kaggle website

On the House Prices Prediction page as illustrated in Fig. 4.10.2, you can find the dataset (under the "Data" tab), submit predictions, see your ranking, etc., The URL is right here:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques



House Prices: Advanced Regression Techniques

Predict sales prices and practice feature engineering, RFs, and gradient boosting

5,012 teams · Ongoing

Start here if...

You have some experience with R or Python and machine learning basics. This is a perfect competition for data science students who have completed an online course in machine learning and are looking to expand their skill set before trying a featured competition.

Competition Description

Fig. 4.10.2: House Price Prediction

[83] https://www.kaggle.com

4.10.3 Accessing and Reading the Dataset

Note that the competition data is separated into training and test sets. Each record includes the property value of the house and attributes such as street type, year of construction, roof type, basement condition, etc. The features represent multiple data types. Year of construction, for example, is represented with integers roof type is a discrete categorical feature, other features are represented with floating point numbers. And here is where reality comes in: for some examples, some data is altogether missing with the missing value marked simply as 'na'. The price of each house is included for the training set only (it is a competition after all). You can partition the training set to create a validation set, but you will only find out how you perform on the official test set when you upload your predictions and receive your score. The "Data" tab on the competition tab has links to download the data.

We will read and process the data using `pandas`, an efficient data analysis toolkit[84], so you will want to make sure that you have `pandas` installed before proceeding further. Fortunately, if you are reading in Jupyter, we can install pandas without even leaving the notebook.

```python
# If pandas is not installed, please uncomment the following line:
# !pip install pandas

%matplotlib inline
import d2l
from mxnet import autograd, init, np, npx
from mxnet.gluon import nn
import pandas as pd
npx.set_np()
```

For convenience, we downloaded and saved the Kaggle dataset in the `DATA_URL` website. For other Kaggle competitions, you may need to download them manually.

```python
# Saved in the d2l package for later use
DATA_HUB['kaggle_house_train'] = (
    DATA_URL+'kaggle_house_pred_train.csv',
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')
DATA_HUB['kaggle_house_test'] = (
    DATA_URL+'kaggle_house_pred_test.csv',
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

To load the two CSV (Comma Separated Values) files containing training and test data respectively we use Pandas.

```python
train_data = pd.read_csv(download('kaggle_house_train'))
test_data = pd.read_csv(download('kaggle_house_test'))
```

The training dataset includes $1,460$ examples, 80 features, and 1 label, while the test data contains $1,459$ examples and 80 features.

```python
print(train_data.shape)
print(test_data.shape)
```

[84] http://pandas.pydata.org/pandas-docs/stable/

```
(1460, 81)
(1459, 80)
```

Let's take a look at the first 4 and last 2 features as well as the label (SalePrice) from the first 4 examples:

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

We can see that in each example, the first feature is the ID. This helps the model identify each training example. While this is convenient, it does not carry any information for prediction purposes. Hence we remove it from the dataset before feeding the data into the network.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

4.10.4 Data Preprocessing

As stated above, we have a wide variety of data types. Before we feed it into a deep network, we need to perform some amount of processing. Let's start with the numerical features. We begin by replacing missing values with the mean. This is a reasonable strategy if features are missing at random. To adjust them to a common scale, we rescale them to zero mean and unit variance. This is accomplished as follows:

$$x \leftarrow \frac{x - \mu}{\sigma}. \tag{4.10.1}$$

To check that this transforms x to data with zero mean and unit variance simply calculate $E[(x - \mu)/\sigma] = (\mu - \mu)/\sigma = 0$. To check the variance we use $E[(x - \mu)^2] = \sigma^2$ and thus the transformed variable has unit variance. The reason for "normalizing" the data is that it brings all features to the same order of magnitude. After all, we do not know *a priori* which features are likely to be relevant.

```
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

Next we deal with discrete values. This includes variables such as 'MSZoning'. We replace them by a one-hot encoding in the same manner as how we transformed multiclass classification data into a vector of 0 and 1. For instance, 'MSZoning' assumes the values 'RL' and 'RM'. They map into vectors $(1, 0)$ and $(0, 1)$ respectively. Pandas does this automatically for us.

```
# Dummy_na=True refers to a missing value being a legal eigenvalue, and
# creates an indicative feature for it
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

```
(2919, 331)
```

You can see that this conversion increases the number of features from 79 to 331. Finally, via the values attribute, we can extract the NumPy format from the Pandas dataframe and convert it into MXNet's native ndarray representation for training.

```
n_train = train_data.shape[0]
train_features = np.array(all_features[:n_train].values, dtype=np.float32)
test_features = np.array(all_features[n_train:].values, dtype=np.float32)
train_labels = np.array(train_data.SalePrice.values,
                        dtype=np.float32).reshape(-1, 1)
```

4.10.5 Training

To get started we train a linear model with squared loss. Not surprisingly, our linear model will not lead to a competition winning submission but it provides a sanity check to see whether there is meaningful information in the data. If we cannot do better than random guessing here, then there might be a good chance that we have a data processing bug. And if things work, the linear model will serve as a baseline giving us some intuition about how close the simple model gets to the best reported models, giving us a sense of how much gain we should expect from fancier models.

```
loss = gluon.loss.L2Loss()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    return net
```

With house prices, as with stock prices, we care about relative quantities more than absolute quantities. More concretely, we tend to care more about the relative error $\frac{y-\hat{y}}{y}$ than about the absolute error $y - \hat{y}$. For instance, if our prediction is off by USD 100,000 when estimating the price of a house in Rural Ohio, where the value of a typical house is 125,000 USD, then we are probably doing a horrible job. On the other hand, if we err by this amount in Los Altos Hills, California, this might represent a stunningly accurate prediction (their, the median house price exceeds 4 million USD).

One way to address this problem is to measure the discrepancy in the logarithm of the price estimates. In fact, this is also the official error metric used by the competition to measure the quality of submissions. After all, a small value δ of $\log y - \log \hat{y}$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following loss function:

$$L = \sqrt{\frac{1}{n}\sum_{i=1}^n \left(\log y_i - \log \hat{y}_i\right)^2}. \tag{4.10.2}$$

```
def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the
    # value less than 1 as 1
    clipped_preds = np.clip(net(features), 1, float('inf'))
    return np.sqrt(2 * loss(np.log(clipped_preds), np.log(labels)).mean())
```

Unlike in previous sections, our training functions here will rely on the Adam optimizer (a slight variant on SGD that we will describe in greater detail later). The main appeal of Adam vs vanilla SGD is that the Adam optimizer, despite doing no better (and sometimes worse) given unlimited resources for hyperparameter optimization, people tend to find that it is significantly less sensitive to the initial learning rate. This will be covered in further detail later on when we discuss the details in Chapter 11.

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    # The Adam optimization algorithm is used here
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
        'learning_rate': learning_rate, 'wd': weight_decay})
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

4.10.6 k-Fold Cross-Validation

If you are reading in a linear fashion, you might recall that we introduced k-fold cross-validation in the section where we discussed how to deal with model section (Section 4.4). We will put this to good use to select the model design and to adjust the hyperparameters. We first need a function that returns the i^{th} fold of the data in a k-fold cross-validation procedure. It proceeds by slicing out the i^{th} segment as validation data and returning the rest as training data. Note that this is not the most efficient way of handling data and we would definitely do something much smarter if our dataset was considerably larger. But this added complexity might obfuscate our code unnecessarily so we can safely omit here owing to the simplicity of our problem.

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
```

(continues on next page)

(continued from previous page)

```
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = np.concatenate((X_train, X_part), axis=0)
            y_train = np.concatenate((y_train, y_part), axis=0)
    return X_train, y_train, X_valid, y_valid
```

The training and verification error averages are returned when we train k times in the k-fold cross-validation.

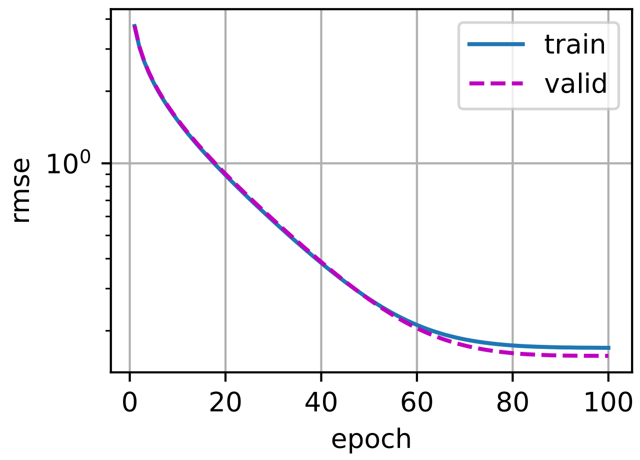
```
def k_fold(k, X_train, y_train, num_epochs,
           learning_rate, weight_decay, batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.plot(list(range(1, num_epochs+1)), [train_ls, valid_ls],
                     xlabel='epoch', ylabel='rmse',
                     legend=['train', 'valid'], yscale='log')
        print('fold %d, train rmse: %f, valid rmse: %f' % (
            i, train_ls[-1], valid_ls[-1]))
    return train_l_sum / k, valid_l_sum / k
```

4.10.7 Model Selection

In this example, we pick an un-tuned set of hyperparameters and leave it up to the reader to improve the model. Finding a good choice can take quite some time, depending on how many things one wants to optimize over. Within reason, the k-fold cross-validation approach is resilient against multiple testing. However, if we were to try out an unreasonably large number of options it might fail since we might just get lucky on the validation split with a particular set of hyperparameters.

```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
      % (k, train_l, valid_l))
```

```
fold 0, train rmse: 0.169805, valid rmse: 0.157211
fold 1, train rmse: 0.162274, valid rmse: 0.189795
fold 2, train rmse: 0.163713, valid rmse: 0.168114
fold 3, train rmse: 0.168033, valid rmse: 0.154878
fold 4, train rmse: 0.163031, valid rmse: 0.182852
5-fold validation: avg train rmse: 0.165371, avg valid rmse: 0.170570
```



You will notice that sometimes the number of training errors for a set of hyper-parameters can be very low, while the number of errors for the K-fold cross-validation may be higher. This is an indicator that we are overfitting. Therefore, when we reduce the amount of training errors, we need to check whether the amount of errors in the k-fold cross-validation have also been reduced accordingly.

4.10.8 Predict and Submit

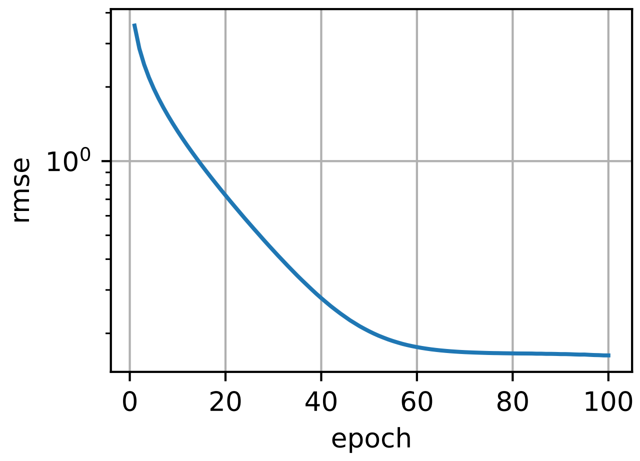
Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/k$ of the data that is used in the cross-validation slices). The model that we obtain in this way can then be applied to the test set. Saving the estimates in a CSV file will simplify uploading the results to Kaggle.

```
def train_and_pred(train_features, test_feature, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
             ylabel='rmse', yscale='log')
    print('train rmse %f' % train_ls[-1])
    # Apply the network to the test set
    preds = net(test_features).asnumpy()
    # Reformat it for export to Kaggle
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

Let's invoke our model. One nice sanity check is to see whether the predictions on the test set resemble those of the k-fold cross-validation process. If they do, it is time to upload them to Kaggle. The following code will generate a file called `submission.csv` (CSV is one of the file formats accepted by Kaggle):

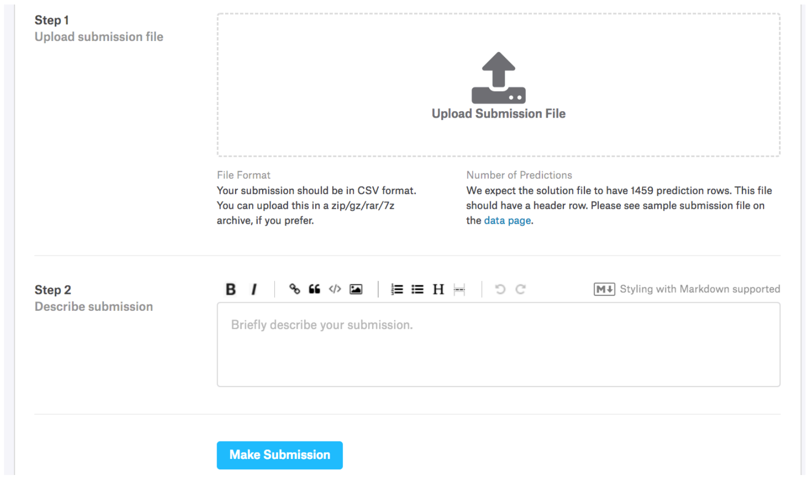
```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train rmse 0.162734
```



Next, as demonstrated in Fig. 4.10.3, we can submit our predictions on Kaggle and see how they compare to the actual house prices (labels) on the test set. The steps are quite simple:

- Log in to the Kaggle website and visit the House Price Prediction Competition page.
- Click the "Submit Predictions" or "Late Submission" button (as of this writing, the button is located on the right).
- Click the "Upload Submission File" button in the dashed box at the bottom of the page and select the prediction file you wish to upload.
- Click the "Make Submission" button at the bottom of the page to view your results.



File Format
Your submission should be in CSV format. You can upload this in a zip/gz/rar/7z archive, if you prefer.

Number of Predictions
We expect the solution file to have 1459 prediction rows. This file should have a header row. Please see sample submission file on the data page.

Briefly describe your submission.

Fig. 4.10.3: Submitting data to Kaggle

Summary

- Real data often contains a mix of different data types and needs to be preprocessed.
- Rescaling real-valued data to zero mean and unit variance is a good default. So is replacing missing values with their mean.
- Transforming categorical variables into indicator variables allows us to treat them like vectors.
- We can use k-fold cross validation to select the model and adjust the hyper-parameters.
- Logarithms are useful for relative loss.

Exercises

1. Submit your predictions for this tutorial to Kaggle. How good are your predictions?
2. Can you improve your model by minimizing the log-price directly? What happens if you try to predict the log price rather than the price?
3. Is it always a good idea to replace missing values by their mean? Hint: can you construct a situation where the values are not missing at random?
4. Find a better representation to deal with missing values. Hint: what happens if you add an indicator variable?
5. Improve the score on Kaggle by tuning the hyperparameters through k-fold cross-validation.
6. Improve the score by improving the model (layers, regularization, dropout).
7. What happens if we do not standardize the continuous numerical features like we have done in this section?