4 | Multilayer Perceptrons

In this chapter, we will introduce your first truly *deep* network. The simplest deep networks are called multilayer perceptrons, and they consist of multiple 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 model selection. To help you combat these problems, we will introduce regularization techniques such as weight decay and dropout. We will also discuss issues relating to numerical stability and parameter initialization that are key to successfully training deep networks. Throughout, we aim to give you a firm grasp not just of the concepts but also of the practice of using deep networks. At the end of this chapter, we apply what we have introduced so far to a real case: house price prediction. We punt matters relating to the computational performance, scalability, and efficiency of our models to subsequent chapters.

4.1 Multilayer Perceptrons

In Chapter 3, we introduced softmax regression (Section 3.4), implementing the algorithm from scratch (Section 3.6) and using high-level APIs (Section 3.7), and training classifiers to recognize 10 categories of clothing from low-resolution images. Along the way, we learned how to wrangle data, coerce our outputs into a valid probability distribution, apply an appropriate loss function, and minimize it with respect to our model's parameters. Now that we have mastered these mechanics in the context of simple linear models, we can launch our exploration of deep neural networks, the comparatively rich class of models with which this book is primarily concerned.

4.1.1 Hidden Layers

We have described the affine transformation in Section 3.1.1, which is a linear transformation added by a bias. To begin, recall the model architecture corresponding to our softmax regression example, illustrated in Fig. 3.4.1. This model mapped our inputs directly to our outputs via a single affine transformation, followed by a softmax operation. If our labels truly were related to our input data by an affine transformation, then this approach would be sufficient. But linearity in affine transformations is a *strong* assumption.

Linear Models May Go Wrong

For example, linearity implies the *weaker* assumption of *monotonicity*: that any increase in our feature must either always cause an increase in our model's output (if the corresponding weight is positive), or always cause a decrease in our model's output (if the corresponding weight is negative). Sometimes that makes sense. For example, if we were trying to predict whether an individual will repay a loan, we might reasonably imagine that holding all else equal, an applicant with a higher income would always be more likely to repay than one with a lower income. While monotonic, this relationship likely is not linearly associated with the probability of repayment. An increase in income from 0 to 50 thousand likely corresponds to a bigger increase in likelihood of repayment than an increase from 1 million to 1.05 million. One way to handle this might be to preprocess our data such that linearity becomes more plausible, say, by using the logarithm of income as our feature.

Note that we can easily come up with examples that violate monotonicity. Say for example that we want to predict probability of death based on body temperature. For individuals with a body temperature above 37°C (98.6°F), higher temperatures indicate greater risk. However, for individuals with body temperatures below 37° C, higher temperatures indicate lower risk! In this case too, we might resolve the problem with some clever preprocessing. Namely, we might use the distance from 37°C as our feature.

But what about classifying images of cats and dogs? Should increasing the intensity of the pixel at location (13, 17) always increase (or always decrease) the likelihood that the image depicts a dog? Reliance on a linear model corresponds to the implicit assumption that the only requirement for differentiating cats vs. dogs is to assess the brightness of individual pixels. This approach is doomed to fail in a world where inverting an image preserves the category.

And yet despite the apparent absurdity of linearity here, as compared with our previous examples, it is less obvious that we could address the problem with a simple preprocessing fix. That is because the significance of any pixel depends in complex ways on its context (the values of the surrounding pixels). While there might exist a representation of our data that would take into account the relevant interactions among our features, on top of which a linear model would be suitable, we simply do not know how to calculate it by hand. With deep neural networks, we used observational data to jointly learn both a representation via hidden layers and a linear predictor that acts upon that representation.

Incorporating Hidden Layers

We can overcome 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 fully-connected layers on top of each other. Each layer feeds into the layer above it, until we generate outputs. We can think of the first $L-1$ layers as our representation and the final layer as our linear predictor. This architecture is commonly called a *multilayer perceptron*, often abbreviated as *MLP*. Below, we depict an MLP diagrammatically (Fig. 4.1.1).

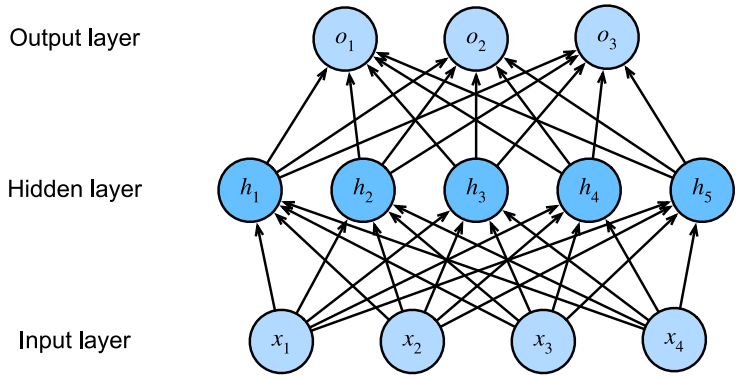


Fig. 4.1.1: An MLP with a hidden layer of 5 hidden units.

This MLP has 4 inputs, 3 outputs, and its hidden layer contains 5 hidden units. Since the input layer does not involve any calculations, producing outputs with this network requires implementing the computations for both the hidden and output layers; thus, the number of layers in this MLP is 2. Note that these layers are both fully connected. Every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer.

From Linear to Nonlinear

As before, by the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, we denote a minibatch of n examples where each example has d inputs (features). For a one-hidden-layer MLP whose hidden layer has h hidden units, denote by $\mathbf{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer, which are *hidden representations*. In mathematics or code, $\mathbf{H}$ is also known as a *hidden-layer variable* or a *hidden variable*. Since the hidden and output layers are both fully connected, we have hidden-layer weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ and output-layer weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. Formally, we calculate the outputs $\mathbf{O} \in \mathbb{R}^{n \times q}$ of the one-hidden-layer MLP as follows:

$$\begin{aligned} \mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.1.1}$$

Note that after adding the hidden layer, our model now requires us to track and update additional sets of parameters. So what have we gained in exchange? You might be surprised to find out that—in the model defined above—*we gain nothing for our troubles*! The reason is plain. The hidden units above are given by an affine function of the inputs, and the outputs (pre-softmax) are just an affine function of the hidden units. An affine function of an affine function is itself an affine function. Moreover, our linear model was already capable of representing any affine function.

We can view the equivalence formally by proving that for any values of the weights, we can just collapse out the hidden layer, yielding an equivalent single-layer model with parameters $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}. \tag{4.1.2}$$

In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear *activation function* σ to be applied to each hidden unit following the affine transformation. The outputs of activation functions (e.g., $\sigma(\cdot)$) are called *activations*. In general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\begin{aligned} \mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.1.3}$$

Since each row in $\mathbf{X}$ corresponds to an example in the minibatch, with some abuse of notation, we define the nonlinearity σ to apply to its inputs in a rowwise fashion, i.e., one example at a time. Note that we used the notation for softmax in the same way to denote a rowwise operation in Section 3.4.4. Often, as in this section, the activation functions that we apply to hidden layers are not merely rowwise, but elementwise. That means that after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units. This is true for most activation functions.

To build more general MLPs, we can continue stacking such hidden layers, e.g., $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, one atop another, yielding ever more expressive models.

Universal Approximators

MLPs can capture complex interactions among our inputs via their hidden neurons, which depend on the values of each of the inputs. We can easily design hidden nodes to perform arbitrary computation, for instance, basic logic operations on a pair of inputs. Moreover, for certain choices of the activation function, it is widely known that MLPs are universal approximators. Even with a single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function, though actually learning that function is the hard part. You might think of your neural network as being a bit like the C programming language. The language, like any other modern language, is capable of expressing any computable program. But actually coming up with a program that meets your specifications is the hard part.

Moreover, just because a single-hidden-layer network *can* learn any function does not mean that you should try to solve all of your problems with single-hidden-layer networks. In fact, we can approximate many functions much more compactly by using deeper (vs. wider) networks. We will touch upon more rigorous arguments in subsequent chapters.

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

4.1.2 Activation Functions

Activation functions decide whether a neuron should be activated or not by calculating the weighted sum and further adding bias with it. They are differentiable operators to transform input signals to outputs, while most of them add non-linearity. Because activation functions are fundamental to deep learning, let us briefly survey some common activation functions.

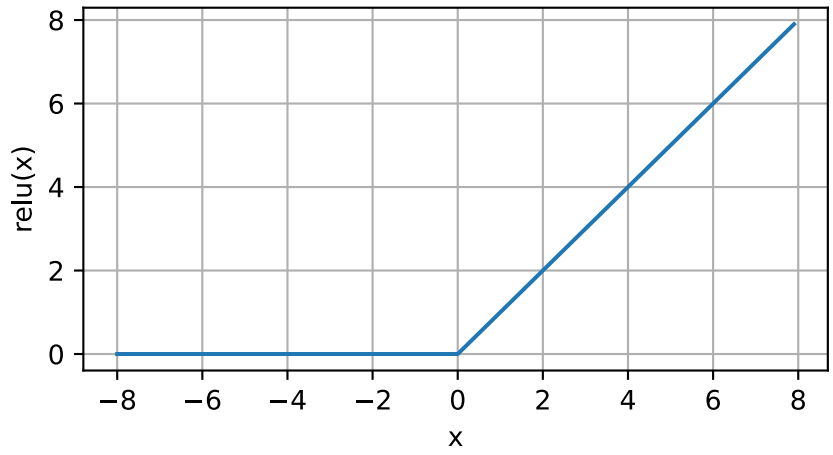
ReLU Function

The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the *rectified linear unit* (*ReLU*). ReLU provides a very simple nonlinear transformation. Given an element x, the function is defined as the maximum of that element and 0:

$$\mathrm{ReLU}(x) = \max(x, 0). \tag{4.1.4}$$

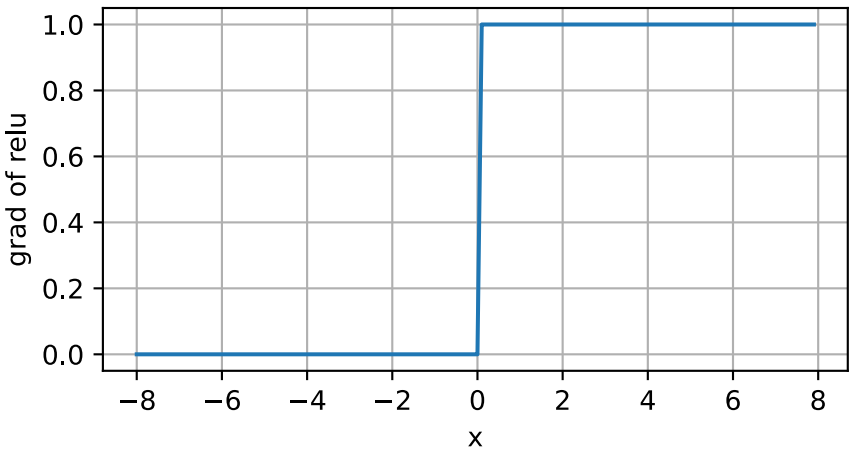
Informally, the ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0. To gain some intuition, we can plot the function. As you can see, the activation function is piecewise linear.

```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.plot(x, y, 'x', 'relu(x)', figsize=(5, 2.5))
```



When the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the 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 default to the left-hand-side 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. We plot the derivative of the ReLU function plotted below.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



The reason for using 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 mitigated the well-documented problem of vanishing gradients that plagued previous versions of neural networks (more on this later).

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

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{4.1.5}$$

Sigmoid Function

The *sigmoid function* transforms its inputs, for which values lie in the domain $\mathbb{R}$, to outputs that lie on 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):

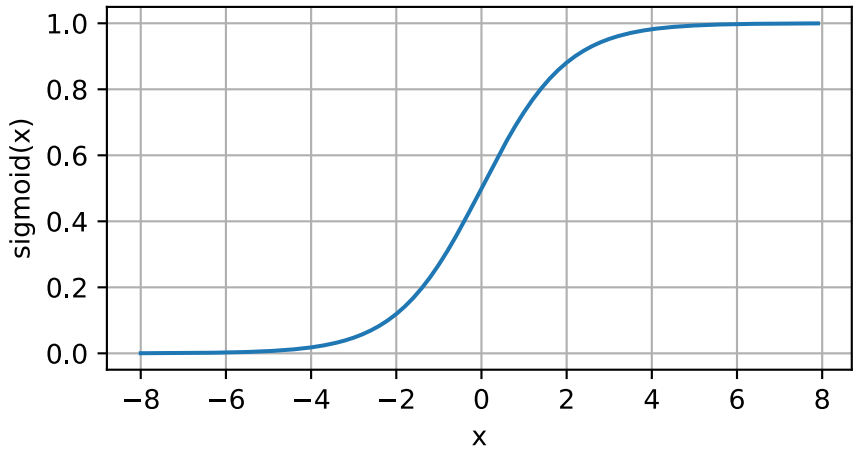
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{4.1.6}$$

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, the inventors of the artificial neuron, focused on thresholding units. A thresholding activation takes value 0 when its input is below some threshold and value 1 when 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 widely used 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). However, the sigmoid has mostly been replaced by the simpler and more easily trainable ReLU for most use in hidden layers. In later chapters on recurrent neural networks, we will describe architectures that leverage sigmoid units to control the flow of information across time.

Below, we plot the sigmoid function. Note that when the input is close to 0, the sigmoid function approaches a linear transformation.

```
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

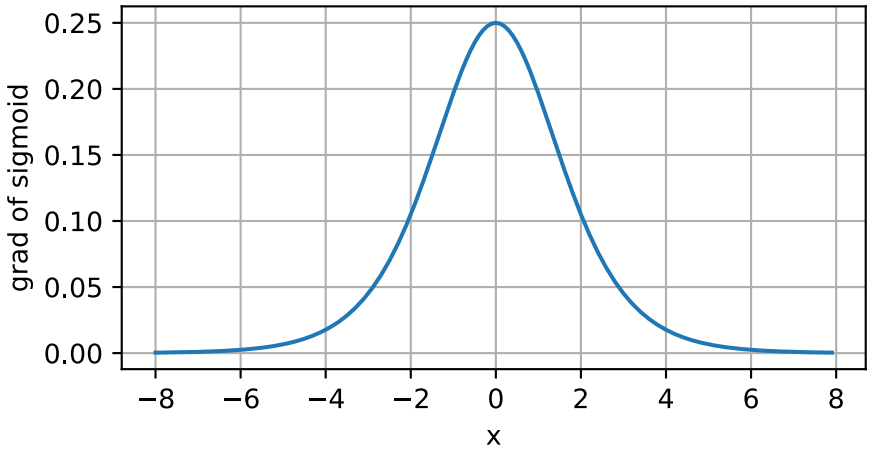


The derivative of the sigmoid function is given by the following equation:

$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\,(1-\text{sigmoid}(x))\,. \tag{4.1.7}$$

The derivative of the 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', figsize=(5, 2.5))
```



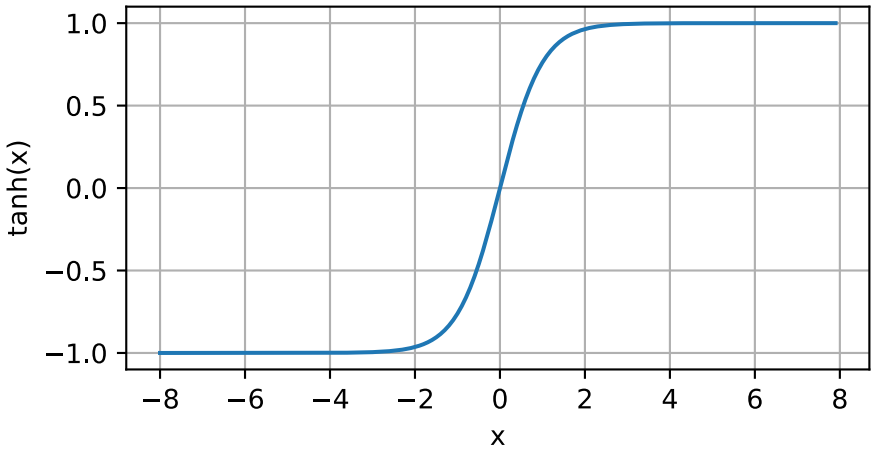
Tanh Function

Like the sigmoid function, the tanh (hyperbolic tangent) function also squashes its inputs, transforming them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}. \tag{4.1.8}$$

We plot the tanh function below. Note that as the input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to that of 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)', figsize=(5, 2.5))
```

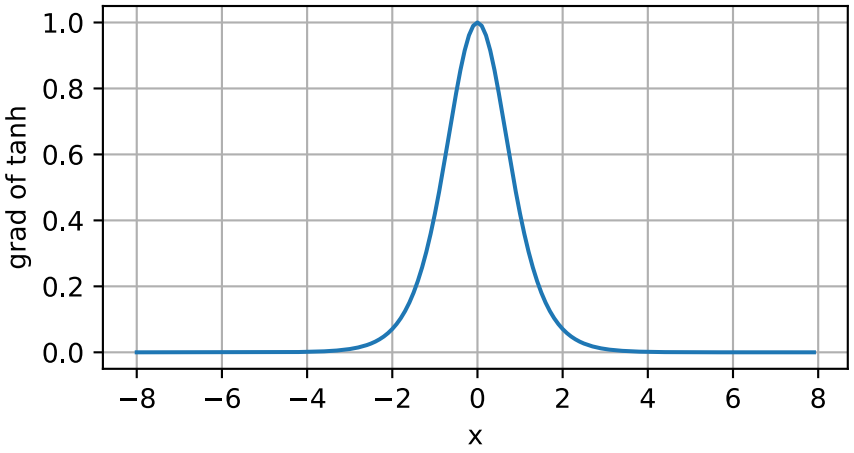


The derivative of the tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{4.1.9}$$

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', figsize=(5, 2.5))
```



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

Summary

- MLP 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 pReLU activation function.
2. Show that an MLP using only ReLU (or pReLU) constructs a continuous piecewise linear function.
3. Show that $\tanh(x) + 1 = 2\,\text{sigmoid}(2x)$.
4. Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems do you expect this to cause?

Discussions[64]

4.2 Implementation of Multilayer Perceptrons from Scratch

Now that we have characterized multilayer perceptrons (MLPs) mathematically, let us try to implement one ourselves. To compare against our previous results achieved with softmax regression (Section 3.6), we will continue to work with the Fashion-MNIST image classification dataset (Section 3.5).

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

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

4.2.1 Initializing Model Parameters

Recall that Fashion-MNIST contains 10 classes, and that each image consists of a $28 \times 28 = 784$ grid of grayscale pixel values. Again, we will disregard the spatial structure among the pixels for now, so we can think of this as simply a classification dataset with 784 input features and 10 classes. To begin, we will implement an MLP with one hidden layer and 256 hidden units. Note that we can regard both of these quantities as hyperparameters. Typically, we choose layer widths in powers of 2, which tend to be computationally efficient because of how memory is allocated and addressed in hardware.

Again, we will represent our parameters with several tensors. Note that *for every layer*, we must keep track of one weight matrix and one bias vector. As always, we allocate memory for the gradients of the loss with respect to these parameters.

[64] https://discuss.d2l.ai/t/90

```
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 implement the ReLU activation ourselves using the maximum function rather than invoking the built-in `relu` function directly.

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

4.2.3 Model

Because we are disregarding spatial structure, we `reshape` each two-dimensional image into a flat vector of length `num_inputs`. Finally, we implement our model with just a few lines of code.

```
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 Loss Function

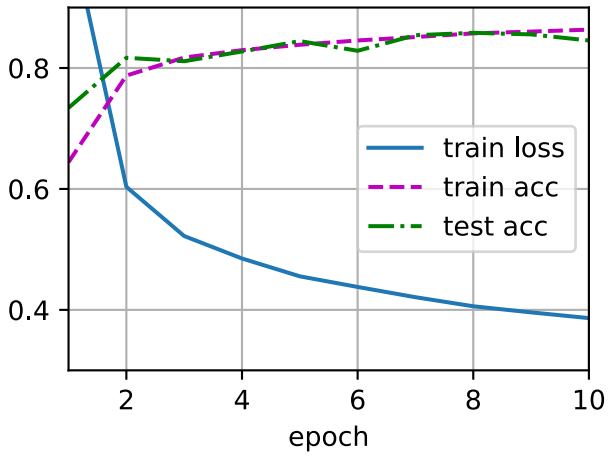
To ensure numerical stability, and because we already implemented the softmax function from scratch (Section 3.6), we leverage the integrated function from high-level APIs for calculating the softmax and cross-entropy loss. Recall our earlier discussion of these intricacies in Section 3.7.2. We encourage the interested reader to examine the source code for the loss function to deepen their knowledge of implementation details.

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

4.2.5 Training

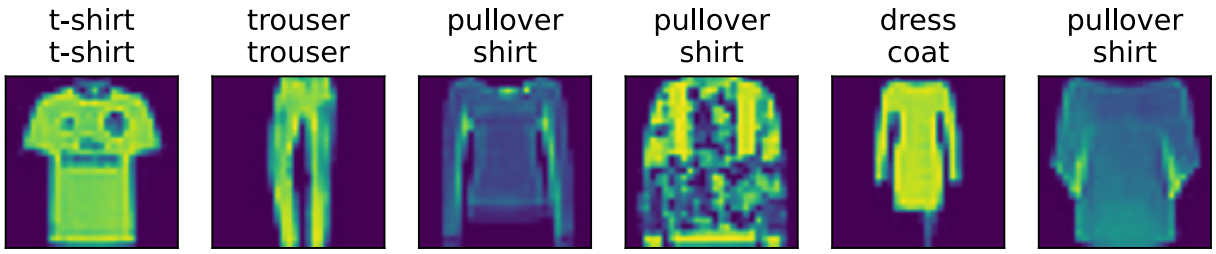
Fortunately, the training loop for MLPs is exactly the same as for softmax regression. Leveraging the `d2l` package again, we call the `train_ch3` function (see Section 3.6), setting the number of epochs to 10 and the learning rate to 0.5.

```
num_epochs, lr = 10, 0.1
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



To evaluate the learned model, we apply it on some test data.

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



Summary

- We saw that implementing a simple MLP is easy, even when done manually.
- However, with a large number of layers, implementing MLPs from scratch can still get messy (e.g., naming and keeping track of our model's parameters).

Exercises

1. Change the value of the hyperparameter `num_hiddens` and see how this hyperparameter influences your results. Determine the best value of this hyperparameter, keeping all others constant.
2. Try adding an additional hidden layer to see how it affects the results.
3. How does changing the learning rate alter your results? Fixing the model architecture and other hyperparameters (including number of epochs), what learning rate gives you the best results?
4. What is the best result you can get by optimizing over all the hyperparameters (learning rate, number of epochs, number of hidden layers, number of hidden units per layer) jointly?
5. Describe why it is much more challenging to deal with multiple hyperparameters.
6. What is the smartest strategy you can think of for structuring a search over multiple hyperparameters?

Discussions[65]

4.3 Concise Implementation of Multilayer Perceptrons

As you might expect, by relying on the high-level APIs, we can implement MLPs even more concisely.

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

4.3.1 Model

As compared with our concise implementation of softmax regression implementation (Section 3.7), the only difference is that we add *two* fully-connected layers (previously, we added *one*). The first is our hidden layer, which contains 256 hidden units and applies the ReLU activation function. The second is our output layer.

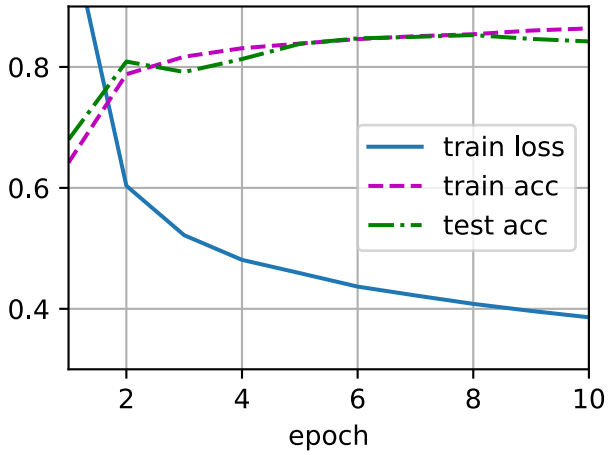
```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

The training loop is exactly the same as when we implemented softmax regression. This modularity enables us to separate matters concerning the model architecture from orthogonal considerations.

[65] https://discuss.d2l.ai/t/92

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
```

```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Summary

- Using high-level APIs, we can implement MLPs much more concisely.
- For the same classification problem, the implementation of an MLP is the same as that of softmax regression except for additional hidden layers with activation functions.

Exercises

1. Try adding different numbers of hidden layers (you may also modify the learning rate). What setting works best?
2. Try out different activation functions. Which one works best?
3. Try different schemes for initializing the weights. What method works best?

Discussions[66]

[66] https://discuss.d2l.ai/t/94

4.4 Model Selection, Underfitting, and Overfitting

As machine learning scientists, our goal is to discover *patterns*. But how can we be sure that we have truly discovered a *general* pattern and not simply memorized our data? For example, imagine that we wanted to hunt for patterns among genetic markers linking patients to their dementia status, where the labels are drawn from the set $\{\text{dementia}, \text{mild cognitive impairment}, \text{healthy}\}$. Because each person's genes identify them uniquely (ignoring identical siblings), it is possible to memorize the entire dataset.

We do not want our model to say *"That's Bob! I remember him! He has dementia!"* The reason why is simple. When we deploy the model in the future, we will encounter patients that the model has never seen before. Our predictions will only be useful if our model has truly discovered a *general* pattern.

To recapitulate more formally, our goal is to discover patterns that capture regularities in the underlying population from which our training set was drawn. If we are successful 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. More often, we must learn from only thousands or tens of thousands of data points. In a large hospital system, we might access hundreds of thousands of medical records. When working with finite samples, we run the risk that we might discover apparent associations that turn out not to hold up when we collect more data.

The phenomenon 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 hyperparameters during the experiment, you might have noticed that with enough neurons, 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 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 her 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 grayscale values, then there are 256^{784} possible images. That means that there are far more low-resolution grayscale thumbnail-sized images than there are atoms in the universe. Even if we could encounter such 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. Suppose that the coin is fair. No matter what algorithm we come up with, 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 algorithm 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.

In the standard supervised learning setting, which we have addressed up until now and will stick with throughout most of this book, we assume that both the training data and the test data are drawn *independently* from *identical* distributions. This is commonly called the *i.i.d. assumption*, which means that the process that samples our data has no memory. In other words, the second example drawn and the third drawn are no more correlated than the second and the two-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 Medical Center, 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

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

some minor violation of the i.i.d. assumption, and yet we have many useful tools for various applications such as face recognition, speech recognition, and language translation.

Other violations are sure to cause trouble. Imagine, for example, if we try 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, 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 vex the greatest minds in learning theory.

When we train our models, we attempt to search 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 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.

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 iterations as more complex, and one subject to *early stopping* (fewer training iterations) as less complex.

It can be difficult to compare the complexity among members of substantially different model classes (say, decision trees vs. neural networks). 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 of a scientific theory: a theory is good if it fits data and if there are specific tests that can be used to disprove it. This is important since all statistical estimation is *post hoc*, 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 will 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 overfitting.
3. The number of training examples. It is 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 MLPs, 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 dataset.

Validation Dataset

In principle we should not touch our test set until after we have chosen all our hyperparameters. 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.

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 common practice to address this problem is to split our data three ways, incorporating a *validation dataset* (or *validation set*) in addition to the training and test datasets. 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 of the book is really the validation accuracy and not a true test set accuracy.

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 errors 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 give 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 model's weights are given by w_i, 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 underfitting vs. overfitting in Fig. 4.4.1.

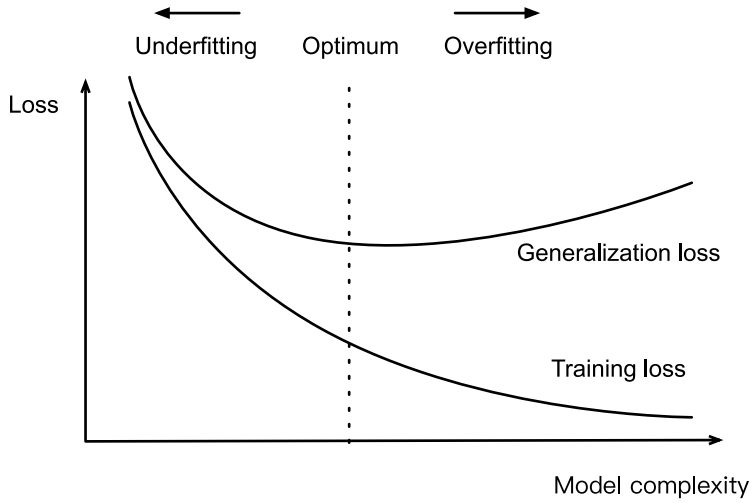


Fig. 4.4.1: Influence of model complexity on underfitting and overfitting

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 hurt. 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 more 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.

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
import math
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:

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

The noise term ϵ obeys a normal distribution with a mean of 0 and a standard deviation of 0.1. For optimization, we typically want to avoid very large values of gradients or losses. This is why the *features* are rescaled from x^i to $\frac{x^i}{i!}$. It allows us to avoid very large values for large exponents i. We will synthesize 100 samples each for the training set and test set.

```
max_degree = 20  # Maximum degree of the polynomial
n_train, n_test = 100, 100  # Training and test dataset sizes
true_w = np.zeros(max_degree)  # 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))
np.random.shuffle(features)
poly_features = np.power(features, np.arange(max_degree).reshape(1, -1))
for i in range(max_degree):
    poly_features[:, i] /= math.gamma(i + 1)  # `gamma(n)` = (n-1)!
# Shape of `labels`: (`n_train` + `n_test`,)
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

Again, monomials stored in `poly_features` are rescaled by the gamma function, where $\Gamma(n) = (n-1)!$. 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([[2.2122064],
        [1.1630787]]),
 array([[1.00000000e+00, 2.21220636e+00, 2.44692850e+00, 1.80437028e+00,
         9.97909844e-01, 4.41516489e-01, 1.62787601e-01, 5.14456779e-02,
         1.42260585e-02, 3.49677517e-03, 7.73558859e-04, 1.55570160e-04,
         2.86794402e-05, 4.88037267e-06, 7.71170789e-07, 1.13732597e-07,
         1.57249964e-08, 2.04629069e-09, 2.51489857e-10, 2.92814419e-11],
        [1.00000000e+00, 1.16307867e+00, 6.76375985e-01, 2.62226164e-01,
         7.62474164e-02, 1.77363474e-02, 3.43812793e-03, 5.71259065e-04,
         8.30524004e-05, 1.07329415e-05, 1.24832559e-06, 1.31990987e-07,
         1.27929916e-08, 1.14455811e-09, 9.50865081e-11, 7.37287228e-12,
         5.35951925e-13, 3.66678970e-14, 2.36931378e-15, 1.45036750e-16]]),
 array([9.629796, 5.51997 ]))
```

Training and Testing the Model

Let us first implement a function to evaluate the loss on a given dataset.

```
def evaluate_loss(net, data_iter, loss):  #@save
    """Evaluate the loss of a model on the given dataset."""
    metric = d2l.Accumulator(2)  # Sum of losses, no. of examples
    for X, y in data_iter:
        l = loss(net(X), y)
        metric.add(d2l.reduce_sum(l), l.size)
    return metric[0] / metric[1]
```

Now define the training function.

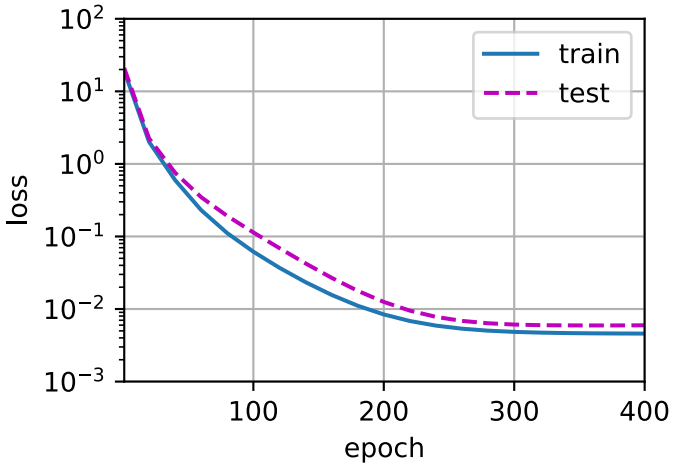
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=400):
    loss = gluon.loss.L2Loss()
    net = nn.Sequential()
    # Switch off the bias since we already catered for it in the polynomial
    # features
    net.add(nn.Dense(1, use_bias=False))
    net.initialize()
    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(num_epochs):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch == 0 or (epoch + 1) % 20 == 0:
            animator.add(epoch + 1, (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, which is the same order as that of the data generation function. The results show that this model's training and test losses can be both effectively reduced. The learned 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^2/2!, x^3/3! from the
# polynomial features
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 4.998419   1.2171801 -3.3890183  5.6006956]]
```

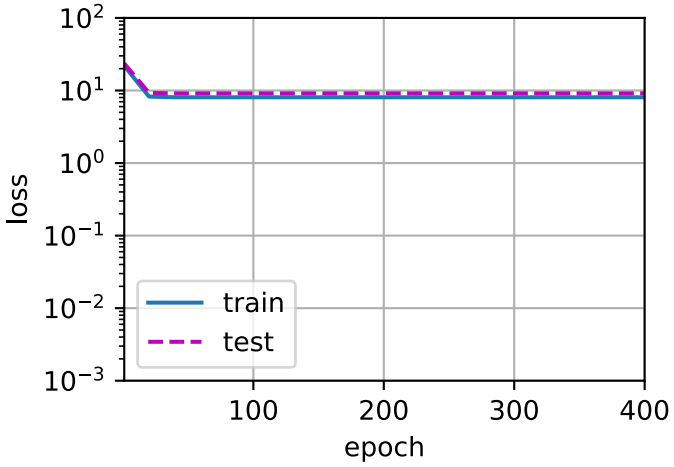


Linear Function Fitting (Underfitting)

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

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

```
weight: [[2.6286998 4.698787 ]]
```

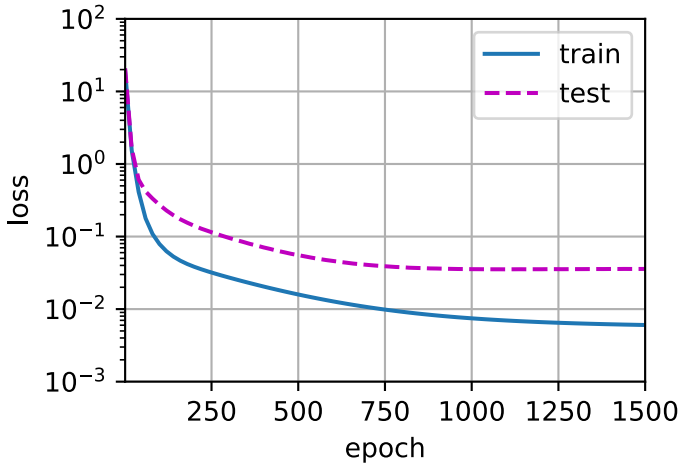


Higher-Order Polynomial Function Fitting (Overfitting)

Now let us try to train the model using a polynomial of too high degree. Here, there are insufficient data to learn that the higher-degree coefficients should have values close to zero. As a result, our overly-complex model is so susceptible that it is being influenced by noise in the training data. Though the training loss can be effectively reduced, the test loss is still much higher. It shows that the complex model overfits the data.

```
# Pick all the dimensions from the polynomial features
train(poly_features[:n_train, :], poly_features[n_train:, :],
      labels[:n_train], labels[n_train:], num_epochs=1500)
```

```
weight: [[ 4.9748263   1.342102   -3.316707    5.018786   -0.1572338   1.4567666
   0.1706778   0.21786575  0.06556528 -0.01586925  0.00981885 -0.05114863
  -0.02413981 -0.01499512 -0.04940709  0.06389925 -0.04761838 -0.04380166
  -0.05188227  0.05655775]]
```



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

Summary

- Since the generalization error cannot be estimated based on the training error, simply minimizing the training error will not necessarily mean a reduction in the generalization error. Machine learning models need to be careful to safeguard against overfitting so 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 a model is not able to reduce the training error. When training error is much lower than validation error, there is overfitting.
- 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. Consider model selection for polynomials:
   - Plot the training loss vs. model complexity (degree of the polynomial). What do you observe? What degree of polynomial do you need to reduce the training loss to 0?
   - Plot the test loss in this case.
   - Generate the same plot as a function of the amount of data.
3. What happens if you drop the normalization ($1/i!$) of the polynomial features x^i? Can you fix this in some other way?
4. Can you ever expect to see zero generalization error?

Discussions[69]

4.5 Weight Decay

Now that we have characterized the problem of overfitting, we can introduce some standard techniques for regularizing models. Recall that we can always mitigate overfitting by going out and collecting more training data. That can be costly, time consuming, or entirely out of our control, making it impossible in the short run. For now, we can assume that we already have as much high-quality data as our resources permit and focus on regularization techniques.

Recall that in our polynomial regression example (Section 4.4) we could limit our model's capacity simply by tweaking the degree of the fitted polynomial. Indeed, limiting the number of features is a popular technique to mitigate overfitting. However, simply tossing aside features can be too blunt an instrument for the job. Sticking with the polynomial regression example, consider what might happen with high-dimensional inputs. The natural extensions of polynomials to multivariate data are called *monomials*, which are simply products of powers of variables. The degree of a monomial is the sum of the powers. For example, $x_1^2 x_2$, and $x_3 x_5^2$ are both monomials of degree 3.

Note that the number of terms with degree d blows up rapidly as d grows larger. Given k variables, the number of monomials of degree d (i.e., k multichoose d) is $\binom{k-1+d}{k-1}$. Even small changes in degree, say from 2 to 3, dramatically increase the complexity of our model. Thus we often need a more fine-grained tool for adjusting function complexity.

4.5.1 Norms and Weight Decay

We have described both the L_2 norm and the L_1 norm, which are special cases of the more general L_p norm in Section 2.3.10. *Weight decay* (commonly called L_2 regularization), might be the most widely-used technique for regularizing parametric machine learning models. The technique is motivated by the basic intuition that among all functions f, the function $f = 0$ (assigning the value 0 to all inputs) is in some sense the *simplest*, and that we can measure the complexity of a function by its distance from 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,

[69] https://discuss.d2l.ai/t/96

including parts of functional analysis and the theory of Banach spaces, are devoted to answering this issue.

One simple interpretation might be to measure the complexity of a linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ by some norm of its weight vector, e.g., $\|\mathbf{w}\|^2$. The most common method for ensuring a small weight vector is to add its norm as a penalty term to the problem of minimizing the loss. Thus we replace our original objective, *minimizing the prediction loss on the training labels*, with new objective, *minimizing the sum of the prediction loss and the penalty term*. Now, if our weight vector grows too large, our learning algorithm might focus on minimizing the weight norm $\|\mathbf{w}\|^2$ vs. minimizing the training error. That is exactly what we want. To illustrate things in code, let us 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 features, $y^{(i)}$ are labels for all data points i, and $(\mathbf{w}, b)$ are the weight and bias parameters, respectively. To penalize the size of the weight vector, we must somehow add $\|\mathbf{w}\|^2$ to the loss function, but how should the model trade off the standard loss for this new additive penalty? In practice, we characterize this tradeoff via the *regularization constant* λ, a non-negative hyperparameter that we fit using validation data:

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

For $\lambda = 0$, we recover our original loss function. For $\lambda > 0$, we restrict the size of $\|\mathbf{w}\|$. 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. The astute reader might wonder why we work with the squared norm and not the standard norm (i.e., the Euclidean distance). We do this for computational convenience. By squaring the L_2 norm, we remove the square root, leaving the sum of squares of each component of the weight vector. This makes the derivative of the penalty easy to compute: the sum of derivatives equals the derivative of the sum.

Moreover, you might ask why we work with the L_2 norm in the first place and not, say, the L_1 norm. In fact, other choices are valid and popular throughout statistics. While L_2-regularized linear models constitute the classic *ridge regression* algorithm, L_1-regularized linear regression is a similarly fundamental model in statistics, which is popularly known as *lasso regression*.

One reason to work with the L_2 norm is that it places an outsize penalty on large components of the weight vector. This biases our learning algorithm towards models that distribute weight evenly across a larger number of features. In practice, this might make them more robust to measurement error in a single variable. By contrast, L_1 penalties lead to models that concentrate weights on a small set of features by clearing the other weights to zero. This is called *feature selection*, which may be desirable for other reasons.

Using the same notation in (3.1.10), the minibatch stochastic gradient descent updates for L_2-regularized regression follow:

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

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 zero. That is why the method is sometimes called "weight decay": given the penalty term alone, our optimization algorithm *decays* the weight at each step of training. In contrast to feature selection, weight decay offers us a continuous mechanism

for adjusting the complexity of a function. Smaller values of λ correspond to less constrained $\mathbf{w}$, whereas larger values of λ constrain $\mathbf{w}$ more considerably.

Whether we include a corresponding bias penalty b^2 can vary across implementations, and may vary across layers of a neural network. Often, we do not regularize the bias term of a network's output layer.

4.5.2 High-Dimensional Linear Regression

We can illustrate the benefits of weight decay through a simple synthetic example.

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

First, we generate some data as before

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2). \tag{4.5.4}$$

We choose our label to be a linear function of our inputs, corrupted by Gaussian noise with zero mean and standard deviation 0.01. To make the effects of overfitting pronounced, we can increase the dimensionality of our problem to $d = 200$ and work with a small training set containing only 20 examples.

```
n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
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

In the following, we will implement weight decay from scratch, simply by adding the squared L_2 penalty to the original target function.

Initializing Model Parameters

First, we will define a function to randomly initialize our model parameters.

```
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.

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

Defining the Training Loop

The following code fits a model on the training set and evaluates it on the test set. The linear network and the squared loss have not changed since Chapter 3, so we will just import them via `d2l.linreg` and `d2l.squared_loss`. The only change here is that our loss now includes the penalty term.

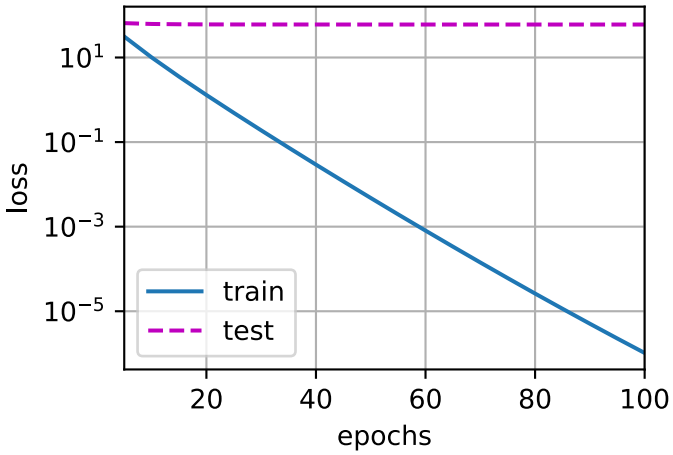
```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                # The L2 norm penalty term has been added, and broadcasting
                # makes `l2_penalty(w)` a vector whose length is `batch_size`
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', np.linalg.norm(w))
```

Training without Regularization

We now run this code with `lambd = 0`, disabling weight decay. Note that we overfit badly, decreasing the training error but not the test error—a textook case of overfitting.

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

```
L2 norm of w: 13.259391
```

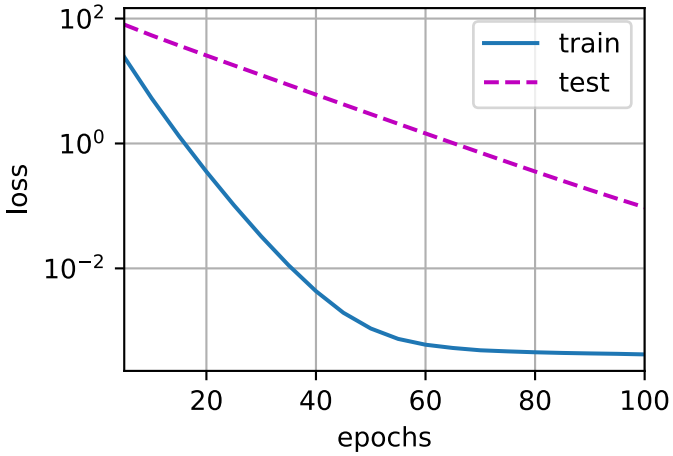


Using Weight Decay

Below, we run with substantial weight decay. Note that the training error increases but the test error decreases. This is precisely the effect we expect from regularization.

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

```
L2 norm of w: 0.382491
```



4.5.4 Concise Implementation

Because weight decay is ubiquitous in neural network optimization, the deep learning framework 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, the optimizer must touch each parameter once anyway.

In the following code, we specify the weight decay hyperparameter directly through `wd` when instantiating our `Trainer`. By default, Gluon decays both weights and biases simultaneously. Note

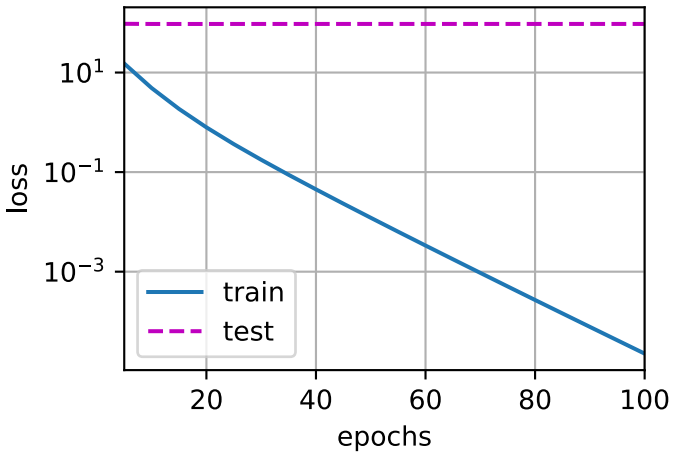
that the hyperparameter `wd` will be multiplied by `wd_mult` when updating model parameters. Thus, if we set `wd_mult` to zero, the bias parameter b will not decay.

```
def train_concise(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
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr, 'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
    net.collect_params('.*bias').setattr('wd_mult', 0)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    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)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', np.linalg.norm(net[0].weight.data()))
```

The plots look identical to those when we implemented weight decay from scratch. However, they run appreciably faster and are easier to implement, a benefit that will become more pronounced for larger problems.

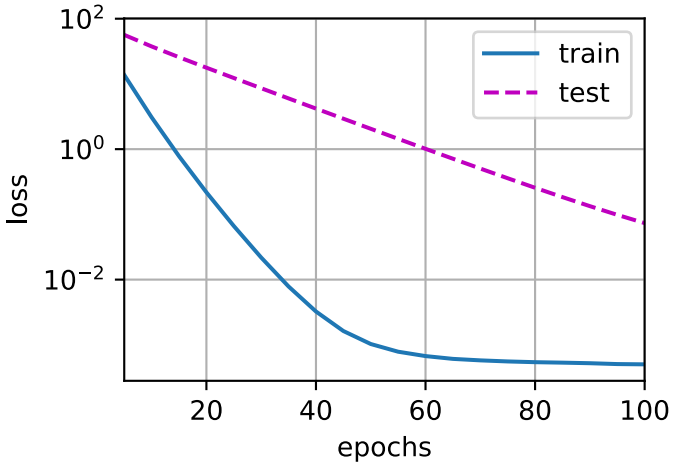
```
train_concise(0)
```

```
L2 norm of w: 15.01407
```




```
train_concise(3)
```

```
L2 norm of w: 0.3399175
```



So far, we only touched upon one notion of what constitutes a simple linear function. Moreover, what constitutes a simple nonlinear function can be an even more complex question. For instance, reproducing kernel Hilbert space (RKHS)[70] allows one to apply tools introduced for linear functions in a nonlinear context. Unfortunately, RKHS-based algorithms tend to scale purely to large, high-dimensional data. In this book we will default to the simple heuristic of applying weight decay on all layers of a deep network.

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.
- The weight decay functionality is provided in optimizers from deep learning frameworks.
- Different sets of parameters can have different update behaviors within the same training loop.

Exercises

1. Experiment with the value of λ in the estimation problem in this section. 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 (L_1 regularization)?
4. We know that $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Can you find a similar equation for matrices (see the Frobenius norm in Section 2.3.10)?

[70] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space

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?

Discussions[71]

4.6 Dropout

In Section 4.5, we introduced the classical approach to 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 zero. More intuitively, we might argue that we encouraged the model to spread out its weights among many features rather than depending too much on a small number of potentially spurious associations.

4.6.1 Overfitting Revisited

Faced with more features than examples, linear models tend to overfit. But given 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. Naively applied, linear models do not take into account interactions among features. For every feature, a linear model must assign either a positive or a negative weight, ignoring context.

In traditional texts, this fundamental tension between generalizability and flexibility is described as the *bias-variance tradeoff*. Linear models have high bias: they can only represent a small class of functions. However, these models have low variance: they give similar results across different random samples of the data.

Deep neural networks inhabit the opposite end of the bias-variance spectrum. Unlike linear models, neural networks are not confined to looking at each feature individually. 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 separately they do not.

Even when we have far more examples than features, deep neural networks are capable of overfitting. In 2017, a group of researchers demonstrated the extreme flexibility of neural networks by training deep nets on randomly-labeled images. Despite the absence of any true pattern linking the inputs to the outputs, they found that the neural network optimized by stochastic gradient descent 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 do better than 10% accuracy on holdout data. The generalization gap here is a whopping 90%. If our models are so expressive that they can overfit this badly, then when should we expect them not to overfit?

The mathematical foundations for the puzzling generalization properties of deep networks remain open research questions, and we encourage the theoretically-oriented reader to dig deeper into the topic. For now, we turn to the investigation of practical tools that tend to empirically improve the generalization of deep nets.

[71] https://discuss.d2l.ai/t/98

4.6.2 Robustness through Perturbations

Let us think briefly about what we expect from a good predictive model. We want it to peform well on unseen data. Classical generalization theory suggests that to close the gap between train and test performance, we should aim for a simple model. Simplicity can come in the form of a small number of dimensions. We explored this when discussing the monomial basis functions of linear models in Section 4.4. Additionally, as we saw when discussing weight decay (L_2 regularization) in Section 4.5, the (inverse) norm of the parameters also represents a useful measure of simplicity. Another useful notion of simplicity is smoothness, i.e., that the function should not be sensitive to small changes to its inputs. 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 this idea when he proved that training with input noise is equivalent to Tikhonov regularization (Bishop, 1995). This work drew a clear mathematical connection between the requirement that a function be smooth (and thus simple), 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 a 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 a deep network with many layers, injecting noise enforces smoothness just on the input-output mapping.

Their idea, called *dropout*, involves injecting noise while computing each internal layer during forward propagation, and it has become a standard technique for training neural networks. The method is called *dropout* because we literally *drop out* some neurons during training. Throughout training, on each iteration, standard dropout consists of zeroing out some fraction of the nodes in each layer before calculating the subsequent layer.

To be clear, we are imposing our own narrative with the link to Bishop. The original paper on dropout offers intuition through a surprising analogy to sexual reproduction. The authors argue that neural network overfitting is characterized by a state in which each layer relies on a specifc pattern of activations in the previous layer, calling this condition *co-adaptation*. Dropout, they claim, breaks up co-adaptation just as sexual reproduction is argued to break up co-adapted genes.

The key challenge then is how to inject this noise. One idea is to inject the noise in an *unbiased* manner so that the expected value of each layer—while fixing the others—equals to the value it would have taken absent noise.

In Bishop's work, he added Gaussian noise to the inputs to a linear model. At each training iteration, he added 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 standard dropout regularization, one debiases each layer by normalizing by the fraction of nodes that were retained (not dropped out). In other words, with *dropout probability* p, each intermediate activation h is replaced by a random variable h' 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., $E[h'] = h$.

4.6.3 Dropout in Practice

Recall the MLP with a hidden layer and 5 hidden units in Fig. 4.1.1. When we apply dropout to a hidden layer, zeroing out each hidden unit with probability p, the result can be viewed 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 the outputs no longer depends on h_2 or h_5 and their respective gradient also vanishes when performing backpropagation. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$.

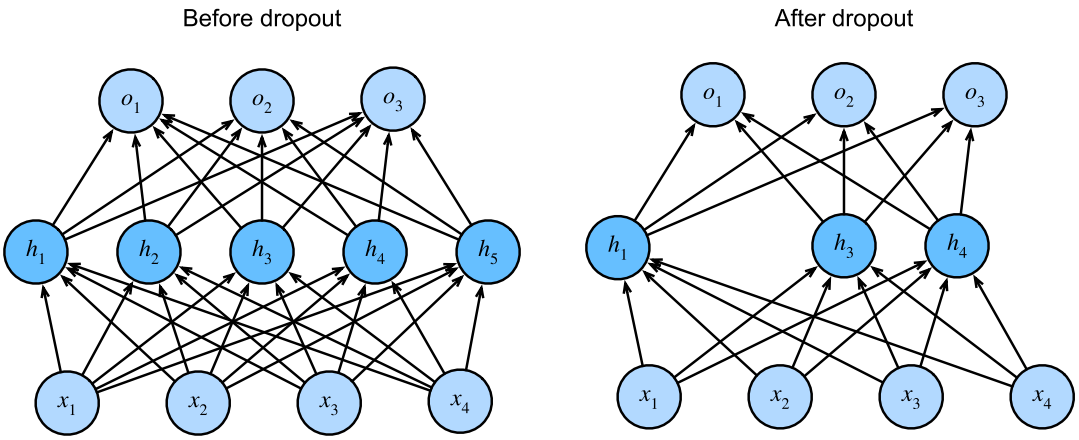


Fig. 4.6.1: MLP before and after dropout.

Typically, we disable dropout at test time. Given a trained model and a new example, we do not drop out any nodes and thus do not need to normalize. However, there are some exceptions: some researchers use dropout at test time as a heuristic for estimating the *uncertainty* of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident.

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_layer` function that drops out the elements in the tensor input `X` with probability `dropout`, rescaling the remainder as described above: dividing the survivors by `1.0-dropout`.

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

def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
```

(continues on next page)

(continued from previous page)

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

We can test out the `dropout_layer` 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_layer(X, 0))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
[[ 0.  2.  4.  6.  8. 10. 12. 14.]
 [ 0. 18. 20.  0.  0.  0. 28.  0.]]
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

Defining Model Parameters

Again, we work with the Fashion-MNIST dataset introduced in Section 3.5. We define an MLP with two hidden layers containing 256 units each.

```
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 below applies dropout to the output of each hidden layer (following the activation function). We can set dropout probabilities for each layer separately. A common trend is 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 layers, respectively. We ensure that dropout is only active during training.

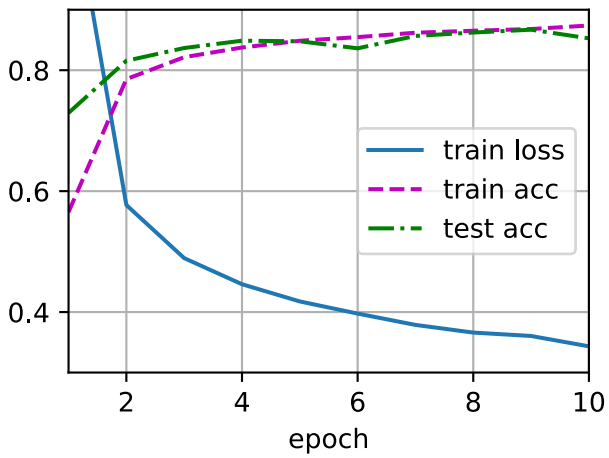
```
dropout1, dropout2 = 0.2, 0.5

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_layer(H1, dropout1)
    H2 = npx.relu(np.dot(H1, W2) + b2)
    if autograd.is_training():
        # Add a dropout layer after the second fully connected layer
        H2 = dropout_layer(H2, dropout2)
    return np.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of MLPs 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))
```



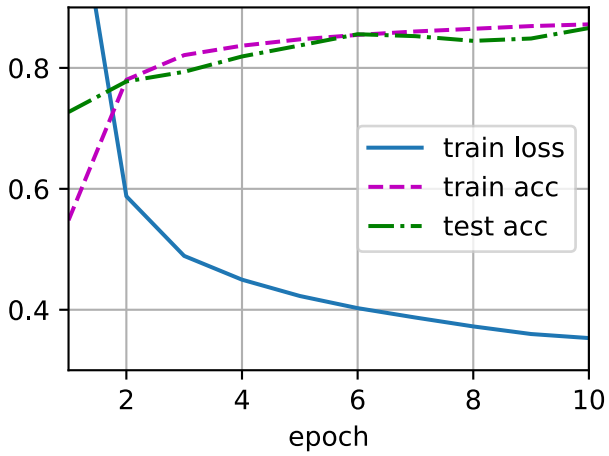
4.6.5 Concise Implementation

With high-level APIs, all we need to do is add a `Dropout` layer 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 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(dropout1),
        nn.Dense(256, activation="relu"),
        # Add a dropout layer after the second fully connected layer
        nn.Dropout(dropout2),
        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 they are used jointly.
- Dropout replaces an activation h with a random variable with expected value h.
- Dropout is only used during training.

Exercises

1. What happens if you change the dropout probabilities for the first and second layers? In particular, what happens if you switch the ones for both layers? Design an experiment to answer these questions, describe your results quantitatively, and summarize the qualitative takeaways.
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. What is the variance of the activations in each hidden layer when dropout is and is not applied? Draw a plot to show how this quantity evolves over time for both models.
4. Why is dropout not typically used at test time?
5. Using the model in this section as an example, compare the effects of using dropout and weight decay. What happens when dropout and weight decay are used at the same time? Are the results additive? Are there diminished returns (or worse)? Do they cancel each other out?
6. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?
7. Invent another technique for injecting random noise at each layer that is different from the standard dropout technique. Can you develop a method that outperforms dropout on the Fashion-MNIST dataset (for a fixed architecture)?

Discussions[72]

4.7 Forward Propagation, Backward Propagation, and Computational Graphs

So far, we have trained our models with minibatch stochastic gradient descent. However, when we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. When it came time to calculate the gradients, we just invoked the backpropagation function provided by the deep learning framework.

The automatic calculation of gradients (automatic differentiation) profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand. Surprisingly often, academic papers had to allocate numerous pages to deriving update rules. While we must continue to rely on automatic differentiation so we can focus on the interesting parts, you ought to know how these gradients are calculated under the hood if you want to go beyond a shallow understanding of deep learning.

In this section, we take a deep dive into the details of *backward propagation* (more commonly called *backpropagation*). To convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs. To start, we focus our exposition on a one-hidden-layer MLP with weight decay (L_2 regularization).

[72] https://discuss.d2l.ai/t/100

4.7.1 Forward Propagation

Forward propagation (or *forward pass*) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. We now work step-by-step through the mechanics of a neural network with one hidden layer. This may seem tedious but in the eternal words of funk virtuoso James Brown, you must "pay the cost to be the boss".

For the sake of simplicity, let us assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and that our hidden layer does not include a bias term. Here the intermediate variable is:

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

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After running the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ through the activation function ϕ we obtain our hidden activation vector of length h,

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

The hidden variable $\mathbf{h}$ is also an intermediate variable. Assuming that 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 of length q:

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

Assuming that 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 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 simply the L_2 norm applied after flattening the matrix into 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* 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, where squares denote variables and circles denote operators. The lower-left corner signifies the input and the upper-right corner is the output. Notice that the directions of the arrows (which illustrate data flow) are primarily rightward and upward.

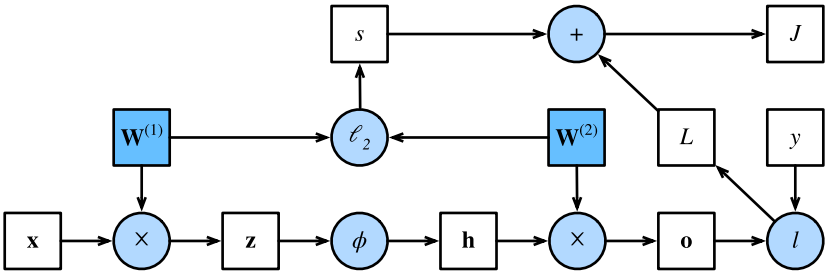


Fig. 4.7.1: Computational graph of forward propagation.

4.7.3 Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y})$, 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 with respect to 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. For higher dimensional tensors, we use the appropriate counterpart. The operator prod hides all the notation overhead.

Recall that the parameters of the simple network with one hidden layer, whose computational graph is in Fig. 4.7.1, 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 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 computational 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 $\mathbf{o}$ 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 to 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 $\mathbf{z}$ 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 Neural Networks

When training neural networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the computational 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.

Take the aforementioned simple network as an example to illustrate. On one hand, computing the regularization term (4.7.5) during forward propagation depends on the current values of model parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. They are given by the optimization algorithm according to backpropagation in the latest iteration. On the other hand, the gradient calculation for the parameter `eq_backprop-J-h` during backpropagation depends on the current value of the hidden variable $\mathbf{h}$, which is given by forward propagation.

Therefore when training neural networks, after model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations. 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 training requires significantly more memory than plain prediction. Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to *out of memory* errors.

Summary

- Forward propagation sequentially calculates and stores intermediate variables within the computational graph defined by the neural network. It proceeds from the input to the output layer.
- Backpropagation 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 than prediction.

Exercises

1. Assume that the inputs $\mathbf{X}$ to some scalar function f are $n \times m$ matrices. What is the dimensionality of the gradient of f with respect to $\mathbf{X}$?
2. Add a bias to the hidden layer of the model described in this section.
    - Draw the corresponding computational graph.
    - Derive the forward and backward propagation equations.
3. Compute the memory footprint for training and prediction in the model described in this section.
4. Assume that you want to compute second derivatives. What happens to the computational graph? How long do you expect the calculation to take?
5. Assume that the computational 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?

Discussions[73]

4.8 Numerical Stability and Initialization

Thus far, every model that we have implemented required that we initialize its parameters according to some pre-specified distribution. Until now, we took the initialization scheme for granted, glossing over the details of how these choices are made. You might have even gotten the impression that these choices are not especially important. To the contrary, the choice of initialization scheme plays a significant role in neural network learning, and it can be crucial for maintaining numerical stability. Moreover, these choices can be tied up in interesting ways with the choice of the nonlinear activation function. Which function we choose and how we initialize parameters can determine how quickly our optimization algorithm converges. Poor choices here can cause us to encounter exploding or vanishing gradients while training. In this section, we delve into these topics with greater detail and discuss some useful heuristics that you will find useful throughout your career in deep learning.

[73] https://discuss.d2l.ai/t/102

4.8.1 Vanishing and Exploding Gradients

Consider a deep network with L layers, input $\mathbf{x}$ and output $\mathbf{o}$. With each layer l defined by a transformation f_l parameterized by weights $\mathbf{W}^{(l)}$, whose hidden variable is $\mathbf{h}^{(l)}$ (let $\mathbf{h}^{(0)} = \mathbf{x}$), our network can be expressed as:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}). \tag{4.8.1}$$

If all the hidden variables and the input are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}^{(l)}$ as follows:

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}. \tag{4.8.2}$$

In other words, this gradient is the product of $L-l$ matrices $\mathbf{M}^{(L)} \cdot \ldots \cdot \mathbf{M}^{(l+1)}$ and the gradient vector $\mathbf{v}^{(l)}$. Thus we are susceptible to the same problems of numerical underflow that often crop up when multiplying together too many probabilities. When dealing with probabilities, a common trick is to switch into log-space, i.e., shifting pressure from the mantissa to the exponent of the numerical representation. Unfortunately, our problem above is more serious: initially the matrices $\mathbf{M}^{(l)}$ may have a wide variety of eigenvalues. They might be small or large, and their product might be *very large* or *very small*.

The risks posed by unstable gradients go beyond numerical representation. Gradients of unpredictable magnitude also threaten the stability of our optimization algorithms. We may be facing parameter updates that are either (i) excessively large, destroying our model (the *exploding gradient* problem); or (ii) excessively small (the *vanishing gradient* problem), rendering learning impossible as parameters hardly move on each update.

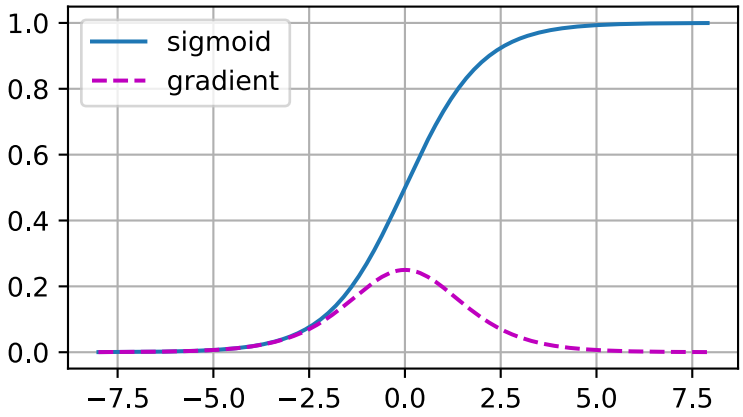
Vanishing Gradients

One frequent culprit causing the vanishing gradient problem is the choice of the activation function σ that is appended following each layer's linear operations. Historically, the sigmoid function $1/(1+\exp(-x))$ (introduced in :numref:sec_mlp) was popular because it resembles a thresholding function. Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that fire either *fully* or *not at all* (like biological neurons) seemed appealing. Let us take a closer look at the sigmoid to see why it can cause vanishing gradients.

```
%matplotlib inline
from d2l import mxnet as 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 you can see, the sigmoid's gradient vanishes both when its inputs are large and when they are small. Moreover, when backpropagating through many layers, unless we are in the Goldilocks zone, where the inputs to many of the sigmoids are close to zero, the gradients of the overall product may vanish. When our network boasts many layers, unless we are careful, the gradient will likely be cut off at some layer. Indeed, this problem used to plague deep network training. Consequently, ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice for practitioners.

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. When this happens due to the initialization of a deep network, we have no 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]]
```

Breaking the Symmetry

Another problem in neural network design is the symmetry inherent in their parametrization. Assume that we have a simple MLP with one hidden layer and two units. 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. Consider the aforementioned one-hidden-layer MLP with two hidden units. For illustration, suppose that the output layer transforms the two hidden units into only one output unit. Imagine what would happen if we initialized all of the parameters of the hidden layer as $\mathbf{W}^{(1)} = c$ for some constant c. In this case, during forward propagation either hidden unit takes the same inputs and parameters, producing the same activation, which is fed to the output unit. During backpropagation, differentiating the output unit with respect to parameters $\mathbf{W}^{(1)}$ gives a gradient whose elements all take the same value. Thus, after gradient-based iteration (e.g., minibatch stochastic gradient descent), all the elements of $\mathbf{W}^{(1)}$ still take the same value. Such iterations would never *break the symmetry* on its own and we might never be able to realize the network's expressive power. The hidden layer would behave as if it had only a single unit. Note that while minibatch stochastic gradient descent 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. Additional care during optimization and suitable regularization can further enhance stability.

Default Initialization

In the previous sections, e.g., in Section 3.3, we used a normal distribution to initialize the values of our weights. If we do not specify the initialization method, the framework will use a default random initialization method, which often works well in practice for moderate problem sizes.

Xavier Initialization

Let us look at the scale distribution of an output (e.g., a hidden variable) o_i for some fully-connected layer *without nonlinearities*. With n_{in} inputs x_j and their associated weights w_{ij} for this layer, an output is given by

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j. \tag{4.8.3}$$

The weights w_{ij} are all drawn independently from the same distribution. Furthermore, let us assume that this distribution has zero mean and variance σ^2. Note that this does not mean that the distribution has to be Gaussian, just that the mean and variance need to exist. For now, let us assume that the inputs to the layer x_j also have zero mean and variance γ^2 and that they are

independent of w_{ij} and independent of each other. In this case, we can compute the mean and variance of o_i as follows:

$$\begin{aligned} E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\ &= 0, \\ \text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\ &= n_{\text{in}} \sigma^2 \gamma^2. \end{aligned} \tag{4.8.4}$$

One way to keep the variance fixed is to set $n_{\text{in}}\sigma^2 = 1$. Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the layers closer to the output. Using the same reasoning as for forward propagation, we see that the gradients' variance can blow up unless $n_{\text{out}}\sigma^2 = 1$, where n_{out} is the number of outputs of this layer. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we simply try to satisfy:

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

This is the reasoning underlying the now-standard and practically beneficial *Xavier initialization*, named after the first author of its creators (Glorot & Bengio, 2010). Typically, the Xavier initialization samples weights from a Gaussian distribution with zero mean and variance $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$. We can also adapt Xavier's intuition to choose the variance when sampling weights from a uniform distribution. Note that the uniform distribution $U(-a, a)$ has variance $\frac{a^2}{3}$. Plugging $\frac{a^2}{3}$ into our condition on σ^2 yields the suggestion to initialize according to

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{4.8.6}$$

Though the assumption for nonexistence of nonlinearities in the above mathematical reasoning can be easily violated in neural networks, the Xavier initialization method turns out to work well in practice.

Beyond

The reasoning above barely scratches the surface of modern approaches to parameter initialization. A deep learning framework often implements over a dozen different heuristics. Moreover, parameter initialization continues to be a hot area of fundamental research in deep learning. Among these are heuristics specialized for tied (shared) parameters, super-resolution, sequence models, and other situations. For instance, Xiao et al. demonstrated the possibility of training 10000-layer neural networks without architectural tricks by using a carefully-designed initialization method (Xiao et al., 2018).

If the topic interests you we suggest a deep dive into this module's offerings, reading the papers that proposed and analyzed each heuristic, and then exploring the latest publications on the topic. Perhaps you will stumble across or even invent a clever idea and contribute an implementation to deep learning frameworks.

Summary

- Vanishing and exploding gradients are common issues in deep networks. Great care in parameter initialization is required to ensure that gradients and parameters remain well controlled.
- Initialization heuristics are needed to ensure that the initial gradients are neither too large nor too small.
- ReLU activation functions mitigate the vanishing gradient problem. This can accelerate convergence.
- Random initialization is key to ensure that symmetry is broken before optimization.
- Xavier initialization suggests that, for each layer, variance of any output is not affected by the number of inputs, and variance of any gradient is not affected by the number of outputs.

Exercises

1. Can you design other cases where a neural network might exhibit symmetry requiring breaking besides the permutation symmetry in an MLP's layers?
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 layer-wise adaptive rate scaling for inspiration (You et al., 2017).

Discussions[74]

4.9 Environment and Distribution Shift

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

Many failed machine learning deployments can be traced back to this pattern. Sometimes models appear to perform marvelously as measured by test set accuracy but fail catastrophically in deployment when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst that perturbs the data distribution. Say, for example, that we trained a model to predict who will repay vs. default on a loan, finding that an applicant's

[74] https://discuss.d2l.ai/t/103

choice of footwear was associated with the 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.

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

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

4.9.1 Types of Distribution Shift

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

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

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

Covariate Shift

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

Consider the challenge of distinguishing cats and dogs. Our training data might consist of images of the kind in Fig. 4.9.1.

Fig. 4.9.1: Training data for distinguishing cats and dogs.

At test time we are asked to classify the images in Fig. 4.9.2.

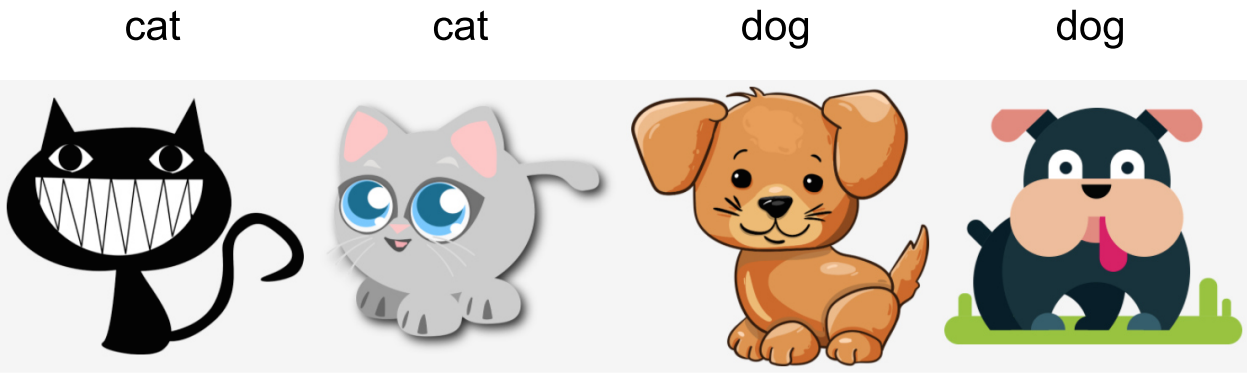


Fig. 4.9.2: Test data for distinguishing cats and dogs.

The training set consists of photos, while the test set contains only cartoons. Training on a dataset with substantially different characteristics from the test set can spell trouble absent a coherent plan for how to adapt to the new domain.

Label Shift

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

Concept Shift

We may also encounter the related problem of *concept shift*, which arises when the very definitions of labels can change. This sounds weird—a *cat* is a *cat*, no? However, other categories are subject to changes in usage over time. Diagnostic criteria for mental illness, what passes for fashionable, and job titles, are all subject to considerable amounts of concept shift. It turns out that if we navigate around the United States, shifting the source of our data by geography, we will find considerable concept shift regarding the distribution of names for *soft drinks* as shown in Fig. 4.9.3.

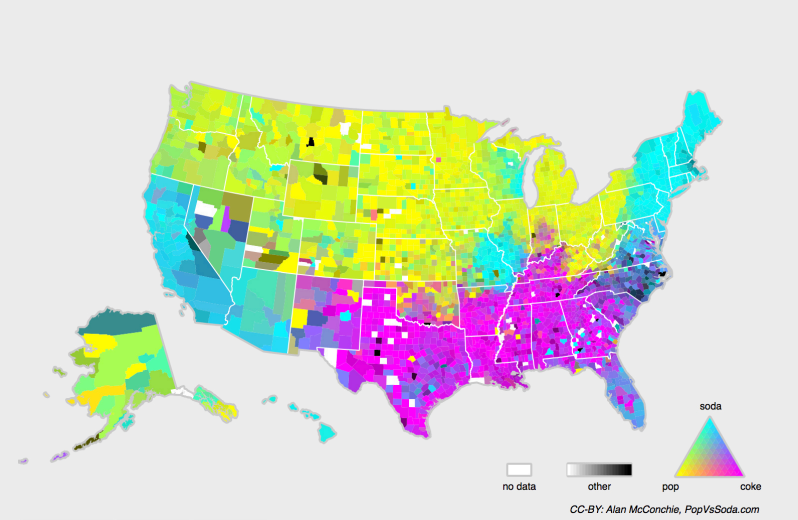


Fig. 4.9.3: Concept shift on soft drink names in the United States.

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

4.9.2 Examples of Distribution Shift

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

Medical Diagnostics

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

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

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

Self-Driving Cars

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

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

Nonstationary Distributions

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

- We train a computational advertising model and then fail to update it frequently (e.g., we forget to incorporate that an obscure new device called an iPad was just launched).
- We build a spam filter. It works well at detecting all spam that we have seen so far. But then the spammers 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 continues to recommend Santa hats long after Christmas.

More Anecdotes

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

4.9.3 Correction of Distribution Shift

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

Empirical Risk and True Risk

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

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i), \tag{4.9.1}$$

where l is the loss function measuring "how bad" the prediction $f(\mathbf{x}_i)$ is given the associated label y_i. Statisticians call the term in (4.9.1) *empirical risk*. *Empirical risk* is an average loss over the training data to approximate the *true risk*, which is the expectation of the loss over the entire population of data drawn from their true distribution $p(\mathbf{x}, y)$:

$$E_{p(\mathbf{x},y)}[l(f(\mathbf{x}), y)] = \int\int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \, d\mathbf{x}dy. \tag{4.9.2}$$

However, in practice we typically cannot obtain the entire population of data. Thus, *empirical risk minimization*, which is minimizing empirical risk in (4.9.1), is a practical strategy for machine learning, with the hope to approximate
minimizing true risk.

Covariate Shift Correction

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

$$\int\int l(f(\mathbf{x}), y) p(y \mid \mathbf{x}) p(\mathbf{x}) \, d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y) q(y \mid \mathbf{x}) q(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} \, d\mathbf{x}dy. \tag{4.9.3}$$

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

$$\beta_i \stackrel{\text{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{4.9.4}$$

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

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i). \tag{4.9.5}$$

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

In this case, there exists a very effective approach that will give almost as good results as the original: logistic regression, which is a special case of softmax regression for binary classification. This is all that is needed to compute estimated probability ratios. We learn a classifier to distinguish between data drawn from $p(\mathbf{x})$ and data drawn from $q(\mathbf{x})$. If it is impossible to distinguish between the two distributions then it means that the associated instances are equally likely to come from either one of 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 $p(\mathbf{x})$ and $q(\mathbf{x})$, respectively. Now denote by z labels that are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{4.9.6}$$

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

$$\beta_i = \frac{1/(1 + \exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1 + \exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{4.9.7}$$

As a result, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a weighted empirical risk minimization problem in (4.9.5) where we weigh terms by β_i.

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

1. Generate a binary-classification training set: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. Train a binary classifier using logistic regression to get function h.
3. Weigh training data using $\beta_i = \exp(h(\mathbf{x}_i))$ or better $\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$ for some constant c.
4. Use weights β_i for training on $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ in (4.9.5).

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

Label Shift Correction

Assume that we are dealing with a classification task with k categories. Using the same notation in Section 4.9.3, q and p are the source distribution (e.g., training time) and target distribution (e.g., test time), respectively. Assume that the distribution of labels shifts over time: $q(y) \neq p(y)$, but the class-conditional distribution stays the same: $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$. If the source distribution $q(y)$ is "wrong", we can correct for that according to the following identity in true risk as defined in (4.9.2):

$$\int \int l(f(\mathbf{x}), y) p(\mathbf{x} \mid y) p(y) \, d\mathbf{x} dy = \int \int l(f(\mathbf{x}), y) q(\mathbf{x} \mid y) q(y) \frac{p(y)}{q(y)} \, d\mathbf{x} dy. \tag{4.9.8}$$

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

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{4.9.9}$$

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

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

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

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

$$\mathbf{C} p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{4.9.10}$$

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

Because we observe the labels on the source data, it is easy to estimate the distribution $q(y)$. Then for any training example i with label y_i, we can take the ratio of our estimated $p(y_i)/q(y_i)$ to calculate the weight β_i, and plug this into weighted empirical risk minimization in (4.9.5).

Concept Shift Correction

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

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

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

4.9.4 A Taxonomy of Learning Problems

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

Batch Learning

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

Online Learning

Now imagine that the data $(\mathbf{x}_i, y_i)$ arrives one sample at a time. More specifically, assume that we first observe $\mathbf{x}_i$, then we need to come up with an estimate $f(\mathbf{x}_i)$ 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. For example, 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, in *online learning*, we have the following cycle where we are continuously improving our model given new observations.

$$\text{model } f_t \longrightarrow \text{data } \mathbf{x}_t \longrightarrow \text{estimate } f_t(\mathbf{x}_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \text{model } f_{t+1} \tag{4.9.11}$$

Bandits

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

Control

In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it will just remember and the response will depend on what happened before. For instance, a coffee boiler controller will observe different temperatures depending on whether it was heating the boiler previously. PID (proportional-integral-derivative) controller algorithms are a popular choice there. Likewise, a user's behavior on a news site will depend on what we showed her previously (e.g., she 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. Recently, control theory (e.g., PID variants) has also been used to automatically tune hyperparameters to achive better disentangling and reconstruction quality, and improve the diversity of generated text and the reconstruction quality of generated images (Shao et al., 2020).

Reinforcement Learning

In the more general case of an environment with memory, we may encounter situations where the environment is trying to cooperate with us (cooperative games, in particular for non-zero-sum games), or others where the environment will try to win. Chess, Go, Backgammon, or StarCraft are some of the cases in *reinforcement learning*. Likewise, we might want to build a good controller for autonomous cars. 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, and trying to cooperate with it.

Considering the Environment

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 she is trying to solve changes over time.

4.9.5 Fairness, Accountability, and Transparency in Machine Learning

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

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

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

Summary

- In many cases training and test sets do not come from the same distribution. This is called distribution shift.
- True risk is the expectation of the loss over the entire population of data drawn from their true distribution. However, this entire population is usually unavailable. Empirical risk is an average loss over the training data to approximate the true risk. In practice, we perform empirical risk minimization.
- Under the corresponding assumptions, covariate and label shift can be detected and corrected for at test time. Failure to account for this bias can become problematic at test time.
- In some cases, the environment may remember automated actions and respond in surprising ways. We must account for this possibility when building models and continue to mon-

itor live systems, open to the possibility that our models and the environment will become entangled in unanticipated ways.

Exercises

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

Discussions[75]

4.10 Predicting House Prices on Kaggle

Now that we have introduced some basic tools for building and training deep networks and regularizing them with techniques including weight decay and dropout, we are ready to put all this knowledge into practice by participating in a Kaggle competition. The house price prediction competition is a great place to start. The data are fairly generic and do not exhibit exotic structure that might require specialized models (as audio or video might). This dataset, collected by Bart de Cock in 2011 (DeCock, 2011), covers house prices in Ames, IA from the period of 2006–2010. It is considerably larger than the famous Boston housing dataset[76] of Harrison and Rubinfeld (1978), boasting both more examples and more features.

In this section, we will walk you through details of data preprocessing, model design, and hyperparameter selection. We hope that through a hands-on approach, you will gain some intuitions that will guide you in your career as a data scientist.

4.10.1 Downloading and Caching Datasets

Throughout the book, we will train and test models on various downloaded datasets. Here, we implement several utility functions to facilitate data downloading. First, we maintain a dictionary `DATA_HUB` that maps a string (the *name* of the dataset) to a tuple containing both the URL to locate the dataset and the SHA-1 key that verifies the integrity of the file. All such datasets are hosted at the site whose address is `DATA_URL`.

```
import os
import requests
import zipfile
import tarfile
import hashlib

DATA_HUB = dict()  #@save
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'  #@save
```

[75] https://discuss.d2l.ai/t/105
[76] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

The following `download` function downloads a dataset, caching it in a local directory (`../data` by default) and returns the name of the downloaded file. If a file corresponding to this dataset already exists in the cache directory and its SHA-1 matches the one stored in `DATA_HUB`, our code will use the cached file to avoid clogging up your internet with redundant downloads.

```
def download(name, cache_dir=os.path.join('..', 'data')):  #@save
    """Download a file inserted into DATA_HUB, return the local filename."""
    assert name in DATA_HUB, f"{name} does not exist in {DATA_HUB}."
    url, sha1_hash = DATA_HUB[name]
    d2l.mkdir_if_not_exist(cache_dir)
    fname = os.path.join(cache_dir, url.split('/')[-1])
    if os.path.exists(fname):
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname  # Hit cache
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname
```

We also implement two additional utility functions: one is to download and extract a zip or tar file and the other to download all the datasets used in this book from `DATA_HUB` into the cache directory.

```
def download_extract(name, folder=None):  #@save
    """Download and extract a zip/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 in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir

def download_all():  #@save
    """Download all files in the DATA_HUB."""
    for name in DATA_HUB:
        download(name)
```

4.10.2 Kaggle

Kaggle[77] is a popular platform that hosts machine learning competitions. Each competition centers on a dataset and many are sponsored by stakeholders who offer prizes to the winning solutions. The platform helps users to interact via forums and shared code, fostering both collaboration and competition. While leaderboard chasing often spirals out of control, with researchers focusing myopically on preprocessing steps rather than asking fundamental questions, there is also tremendous value in the objectivity of a platform that facilitates direct quantitative comparisons among competing approaches as well as code sharing so that everyone can learn what did and did not work. If you want to participate in a Kaggle competitions, you will first need to register for an account (see Fig. 4.10.1).

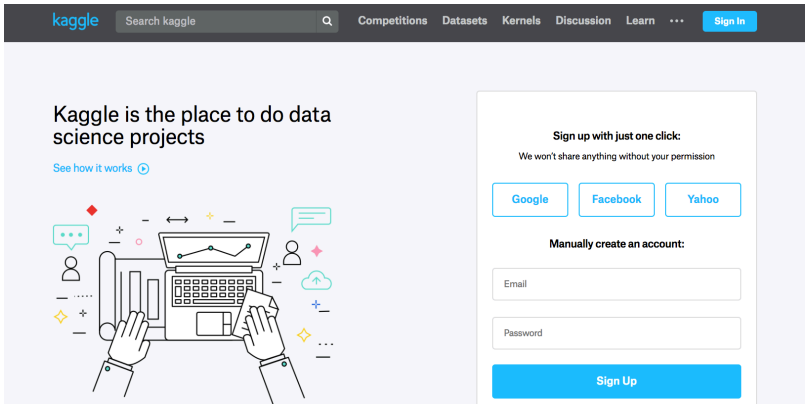


Fig. 4.10.1: The Kaggle website.

On the house price prediction competition page, as illustrated in Fig. 4.10.2, you can find the dataset (under the "Data" tab), submit predictions, and see your ranking, The URL is right here:

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

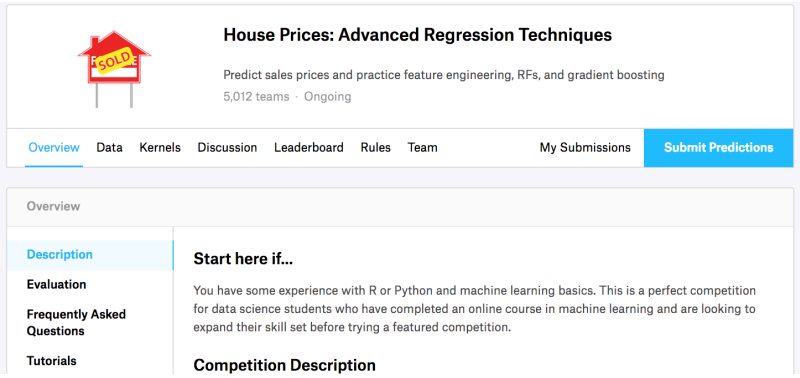


Fig. 4.10.2: The house price prediction competition page.

[77] 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 consist of various data types. For example, the year of construction is represented by an integer, the roof type by discrete categorical assignments, and other features by floating point numbers. And here is where reality complicates things: for some examples, some data are 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). We will want to partition the training set to create a validation set, but we only get to evaluate our models on the official test set after uploading predictions to Kaggle. The "Data" tab on the competition tab in Fig. 4.10.2 has links to download the data.

To get started, we will read in and process the data using pandas, which we have introduced in Section 2.2. 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.

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

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

For convenience, we can download and cache the Kaggle housing dataset using the script we defined above.

```
DATA_HUB['kaggle_house_train'] = (  #@save
    DATA_URL + 'kaggle_house_pred_train.csv',
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')

DATA_HUB['kaggle_house_test'] = (  #@save
    DATA_URL + 'kaggle_house_pred_test.csv',
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

We use pandas to load the two csv files containing training and test data respectively.

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

```
Downloading ../data/kaggle_house_pred_train.csv from http://d2l-data.s3-accelerate.amazonaws.
↪com/kaggle_house_pred_train.csv...
Downloading ../data/kaggle_house_pred_test.csv from http://d2l-data.s3-accelerate.amazonaws.
↪com/kaggle_house_pred_test.csv...
```

The training dataset includes 1460 examples, 80 features, and 1 label, while the test data contains 1459 examples and 80 features.

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

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

Let us take a look at the first four and last two features as well as the label (SalePrice) from the first four 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 model.

```
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. We will need to preprocess the data before we can start modeling. Let us start with the numerical features. First, we apply a heuristic, replacing all missing values by the corresponding feature's mean. Then, to put all features on a common scale, we *standardize* the data by rescaling features to zero mean and unit variance:

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

To verify that this indeed transforms our feature (variable) such that it has zero mean and unit variance, note that $E[\frac{x-\mu}{\sigma}] = \frac{\mu-\mu}{\sigma} = 0$ and that $E[(x-\mu)^2] = (\sigma^2+\mu^2)-2\mu^2+\mu^2 = \sigma^2$. Intuitively, we standardize the data for two reasons. First, it proves convenient for optimization. Second, because we do not know *a priori* which features will be relevant, we do not want to penalize coefficients assigned to one feature more than on any other.

```
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 features such as "MSZoning". We replace them by a one-hot encoding in the same way that we previously transformed multiclass labels into vectors (see Section 3.4.1). For instance, "MSZoning" assumes the values "RL" and "RM". Dropping the "MSZoning" feature, two new indicator features "MSZoning_RL" and "MSZoning_RM" are created

with values being either 0 or 1. According to one-hot encoding, if the original value of "MSZoning" is "RL", then "MSZoning_RL" is 1 and "MSZoning_RM" is 0. The pandas package does this automatically for us.

```
# `Dummy_na=True` considers "na" (missing value) as a valid feature value, and
# creates an indicator 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 format and convert it into the tensor 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.reshape(-1, 1), dtype=np.float32)
```

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. Thus 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 (there, 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 measure used by the competition to evaluate the quality of submissions. After all, a small value δ for $|\log y - \log \hat{y}| \leq \delta$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following root-mean-squared-error between the logarithm of the predicted price and

the logarithm of the label price:

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

```
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 will rely on the Adam optimizer (we will describe it in greater detail later). The main appeal of this optimizer is that, 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.

```
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

You might recall that we introduced K-fold cross-validation in the section where we discussed how to deal with model selection (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 it 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)
```

(continues on next page)

(continued from previous page)

```
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = np.concatenate([X_train, X_part], 0)
            y_train = np.concatenate([y_train, y_part], 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(f'fold {i + 1}, train log rmse {float(train_ls[-1]):f}, '
              f'valid log rmse {float(valid_ls[-1]):f}')
    return train_l_sum / k, valid_l_sum / k
```

4.10.7 Model Selection

In this example, we pick an untuned set of hyperparameters and leave it up to the reader to improve the model. Finding a good choice can take time, depending on how many variables one optimizes over. With a large enough dataset, and the normal sorts of hyperparameters, K-fold cross-validation tends to be reasonably resilient against multiple testing. However, if we try an unreasonably large number of options we might just get lucky and find that our validation performance is no longer representative of the true error.

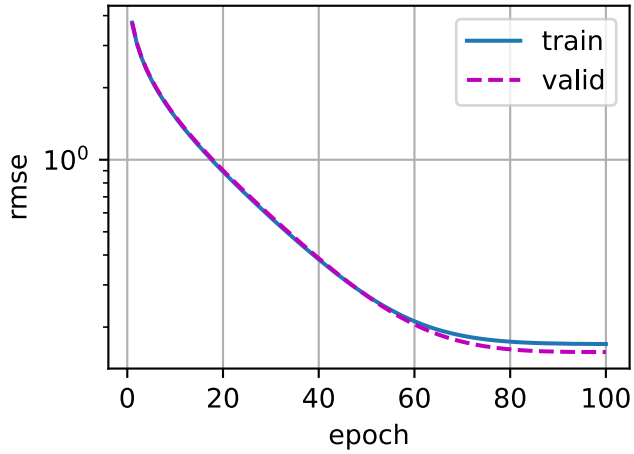
```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print(f'{k}-fold validation: avg train log rmse: {float(train_l):f}, '
      f'avg valid log rmse: {float(valid_l):f}')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

```
fold 1, train log rmse 0.169854, valid log rmse 0.157272
fold 2, train log rmse 0.161957, valid log rmse 0.188484
fold 3, train log rmse 0.163386, valid log rmse 0.167722
fold 4, train log rmse 0.167667, valid log rmse 0.154572
```

(continues on next page)

(continued from previous page)

```
fold 5, train log rmse 0.162538, valid log rmse 0.182737
5-fold validation: avg train log rmse: 0.165080, avg valid log rmse: 0.170157
```



Notice that sometimes the number of training errors for a set of hyperparameters can be very low, even as the number of errors on K-fold cross-validation is considerably higher. This indicates that we are overfitting. Throughout training you will want to monitor both numbers. Less overfitting might indicate that our data can support a more powerful model. Massive overfitting might suggest that we can gain by incorporating regularization techniques.

4.10.8 Submitting Predictions on Kaggle

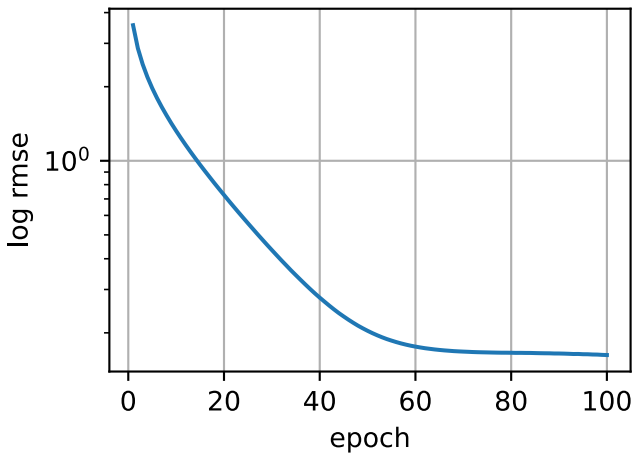
Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/K$ of the data that are used in the cross-validation slices). The model that we obtain in this way can then be applied to the test set. Saving the predictions in 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='log rmse', yscale='log')
    print(f'train log rmse {float(train_ls[-1]):f}')
    # Apply the network to the test set
    preds = d2l.numpy(net(test_features))
    # Reformat it to 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)
```

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`.

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

```
train log rmse 0.162503
```



Next, as demonstrated in Fig. 4.10.3, we can submit our predictions on Kaggle and see how they compare with 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.

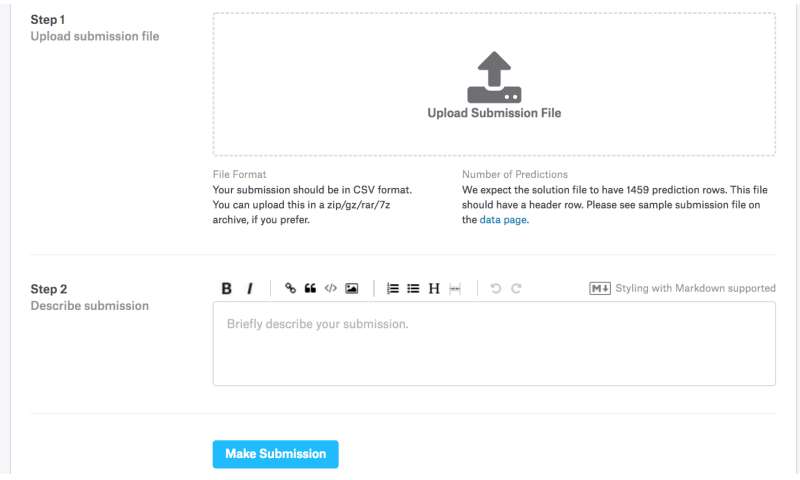


Fig. 4.10.3: Submitting data to Kaggle

Summary

- Real data often contain a mix of different data types and need 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 features into indicator features allows us to treat them like one-hot vectors.
- We can use K-fold cross-validation to select the model and adjust the hyperparameters.
- Logarithms are useful for relative errors.

Exercises

1. Submit your predictions for this section to Kaggle. How good are your predictions?
2. Can you improve your model by minimizing the logarithm of prices directly? What happens if you try to predict the logarithm of the 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. Improve the score on Kaggle by tuning the hyperparameters through K-fold cross-validation.
5. Improve the score by improving the model (e.g., layers, weight decay, and dropout).
6. What happens if we do not standardize the continuous numerical features like what we have done in this section?

Discussions[78]

[78] https://discuss.d2l.ai/t/106