3 | Linear Neural Networks

Before we get into the details of deep neural networks, we need to cover the basics of neural network training. In this chapter, we will cover the entire training process, including defining simple neural network architectures, handling data, specifying a loss function, and training the model. In order to make things easier to grasp, we begin with the simplest concepts. Fortunately, classic statistical learning techniques such as linear and logistic regression can be cast as *shallow* neural networks. Starting from these classic algorithms, we will introduce you to the basics, providing the basis for more complex techniques such as softmax regression (introduced at the end of this chapter) and multilayer perceptrons (introduced in the next chapter).

3.1 Linear Regression

Regression refers to a set of methods for modeling the relationship between data points $\mathbf{x}$ and corresponding real-valued targets y. In the natural sciences and social sciences, the purpose of regression is most often to *characterize* the relationship between the inputs and outputs. Machine learning, on the other hand, is most often concerned with *prediction*.

Regression problems pop up whenever we want to predict a numerical value. Common examples include predicting prices (of homes, stocks, etc.), predicting length of stay (for patients in the hospital), demand forecasting (for retail sales), among countless others. Not every prediction problem is a classic *regression* problem. In subsequent sections, we will introduce classification problems, where the goal is to predict membership among a set of categories.

3.1.1 Basic Elements of Linear Regression

Linear regression may be both the simplest and most popular among the standard tools to regression. Dating back to the dawn of the 19th century, linear regression flows from a few simple assumptions. First, we assume that the relationship between the *features* $\mathbf{x}$ and targets y is linear, i.e., that y can be expressed as a weighted sum of the inputs $\mathbf{x}$, give or take some noise on the observations. Second, we assume that any noise is well-behaved (following a Gaussian distribution). To motivate the approach, let's start with a running example. Suppose that we wish to estimate the prices of houses (in dollars) based on their area (in square feet) and age (in years).

To actually fit a model for predicting house prices, we would need to get our hands on a dataset consisting of sales for which we know the sale price, area and age for each home. In the terminology of machine learning, the dataset is called a *training data* or *training set*, and each row (here the data corresponding to one sale) is called an *instance* or *example*. The thing we are trying to predict (here, the price) is called a *target* or *label*. The variables (here *age* and *area*) upon which the predictions are based are called *features* or *covariates*.

Typically, we will use n to denote the number of examples in our dataset. We index the samples by i, denoting each input data point as $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$ and the corresponding label as $y^{(i)}$.

Linear Model

The linearity assumption just says that the target (price) can be expressed as a weighted sum of the features (area and age):

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{3.1.1}$$

Here, w_{area} and w_{age} are called *weights*, and b is called a *bias* (also called an *offset* or *intercept*). The weights determine the influence of each feature on our prediction and the bias just says what value the predicted price should take when all of the features take value 0. Even if we will never see any homes with zero area, or that are precisely zero years old, we still need the intercept or else we will limit the expressivity of our linear model.

Given a dataset, our goal is to choose the weights w and bias b such that on average, the predictions made according our model best fit the true prices observed in the data.

In disciplines where it is common to focus on datasets with just a few features, explicitly expressing models long-form like this is common. In ML, we usually work with high-dimensional datasets, so it is more convenient to employ linear algebra notation. When our inputs consist of d features, we express our prediction $\hat{y}$ as

$$\hat{y} = w_1 \cdot x_1 + ... + w_d \cdot x_d + b. \tag{3.1.2}$$

Collecting all features into a vector $\mathbf{x}$ and all weights into a vector $\mathbf{w}$, we can express our model compactly using a dot product:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b. \tag{3.1.3}$$

Here, the vector $\mathbf{x}$ corresponds to a single data point. We will often find it convenient to refer to our entire dataset via the *design matrix* X. Here, X contains one row for every example and one column for every feature.

For a collection of data points $\mathbf{X}$, the predictions $\hat{\mathbf{y}}$ can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b. \tag{3.1.4}$$

Given a training dataset X and corresponding (known) targets $\mathbf{y}$, the goal of linear regression is to find the *weight* vector w and bias term b that given some a new data point $\mathbf{x}_i$, sampled from the same distribution as the training data will (in expectation) predict the target y_i with the lowest error.

Even if we believe that the best model for predicting y given $\mathbf{x}$ is linear, we would not expect to find real-world data where y_i exactly equals $\mathbf{w}^T \mathbf{x} + b$ for all points $(\mathbf{x}, y)$. For example, whatever instruments we use to observe the features X and labels $\mathbf{y}$ might suffer small amount of measurement error. Thus, even when we are confident that the underlying relationship is linear, we will incorporate a noise term to account for such errors.

Before we can go about searching for the best parameters w and b, we will need two more things: (i) a quality measure for some given model; and (ii) a procedure for updating the model to improve its quality.

Loss Function

Before we start thinking about how *to fit* our model, we need to determine a measure of *fitness*. The *loss function* quantifies the distance between the *real* and *predicted* value of the target. The loss will usually be a non-negative number where smaller values are better and perfect predictions incur a loss of 0. The most popular loss function in regression problems is the sum of squared errors. When our prediction for some example i is $\hat{y}^{(i)}$ and the corresponding true label is $y^{(i)}$, the squared error is given by:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{3.1.5}$$

The constant $1/2$ makes no real difference but will prove notationally convenient, cancelling out when we take the derivative of the loss. Since the training dataset is given to us, and thus out of our control, the empirical error is only a function of the model parameters. To make things more concrete, consider the example below where we plot a regression problem for a one-dimensional case as shown in Fig. 3.1.1.

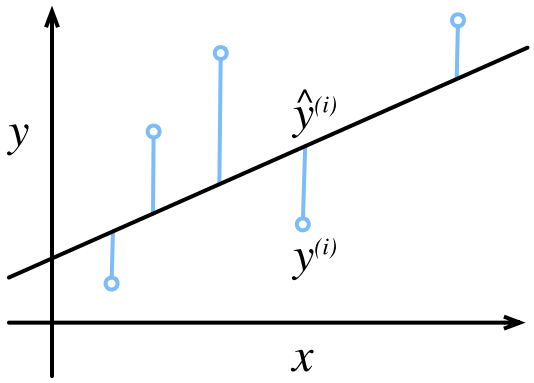


Fig. 3.1.1: Fit data with a linear model.

Note that large differences between estimates $\hat{y}^{(i)}$ and observations $y^{(i)}$ lead to even larger contributions to the loss, due to the quadratic dependence. To measure the quality of a model on the entire dataset, we simply average (or equivalently, sum) the losses on the training set.

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{3.1.6}$$

When training the model, we want to find parameters $(\mathbf{w}^*, b^*)$ that minimize the total loss across all training samples:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b). \tag{3.1.7}$$

Analytic Solution

Linear regression happens to be an unusually simple optimization problem. Unlike most other models that we will encounter in this book, linear regression can be solved analytically by applying a simple formula, yielding a global optimum. To start, we can subsume the bias b into the parameter $\mathbf{w}$ by appending a column to the design matrix consisting of all $1s$. Then our prediction problem is to minimize $||\mathbf{y} - X\mathbf{w}||$. Because this expression has a quadratic form, it is convex, and so long as the problem is not degenerate (our features are linearly independent), it is strictly convex.

Thus there is just one critical point on the loss surface and it corresponds to the global minimum. Taking the derivative of the loss with respect to $\mathbf{w}$ and setting it equal to 0 yields the analytic solution:

$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y. \tag{3.1.8}$$

While simple problems like linear regression may admit analytic solutions, you should not get used to such good fortune. Although analytic solutions allow for nice mathematical analysis, the requirement of an analytic solution is so restrictive that it would exclude all of deep learning.

Gradient descent

Even in cases where we cannot solve the models analytically, and even when the loss surfaces are high-dimensional and nonconvex, it turns out that we can still train models effectively in practice. Moreover, for many tasks, these difficult-to-optimize models turn out to be so much better that figuring out how to train them ends up being well worth the trouble.

The key technique for optimizing nearly any deep learning model, and which we will call upon throughout this book, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called *gradient descent*. On convex loss surfaces, it will eventually converge to a global minimum, and while the same cannot be said for nonconvex surfaces, it will at least lead towards a (hopefully good) local minimum.

The most naive application of gradient descent consists of taking the derivative of the true loss, which is an average of the losses computed on every single example in the dataset. In practice, this can be extremely slow. We must pass over the entire dataset before making a single update. Thus, we will often settle for sampling a random minibatch of examples every time we need to computer the update, a variant called *stochastic gradient descent*.

In each iteration, we first randomly sample a minibatch $\mathcal{B}$ consisting of a fixed number of training data examples. We then compute the derivative (gradient) of the average loss on the mini batch with regard to the model parameters. Finally, we multiply the gradient by a predetermined step size $\eta > 0$ and subtract the resulting term from the current parameter values.

We can express the update mathematically as follows (∂ denotes the partial derivative) :

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b). \tag{3.1.9}$$

To summarize, steps of the algorithm are the following: (i) we initialize the values of the model parameters, typically at random; (ii) we iteratively sample random batches from the the data (many times), updating the parameters in the direction of the negative gradient.

For quadratic losses and linear functions, we can write this out explicitly as follows: Note that $\mathbf{w}$ and $\mathbf{x}$ are vectors. Here, the more elegant vector notation makes the math much more readable than expressing things in terms of coefficients, say $w_1, w_2, \ldots, w_d$.

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) &&= w - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) &&= b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned} \tag{3.1.10}$$

In the above equation, $|\mathcal{B}|$ represents the number of examples in each minibatch (the *batch size*) and η denotes the *learning rate*. We emphasize that the values of the batch size and learning rate are manually pre-specified and not typically learned through model training. These parameters that are tunable but not updated in the training loop are called *hyper-parameters*. *Hyperparameter tuning* is the process by which these are chosen, and typically requires that we adjust the hyper-parameters based on the results of the inner (training) loop as assessed on a separate *validation* split of the data.

After training for some predetermined number of iterations (or until some other stopping criteria is met), we record the estimated model parameters, denoted $\hat{\mathbf{w}}, \hat{b}$ (in general the "hat" symbol denotes estimates). Note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss because, although the algorithm converges slowly towards a local minimum it cannot achieve it exactly in a finite number of steps.

Linear regression happens to be a convex learning problem, and thus there is only one (global) minimum. However, for more complicated models, like deep networks, the loss surfaces contain many minima. Fortunately, for reasons that are not yet fully understood, deep learning practitioners seldom struggle to find parameters that minimize the loss *on training data*. The more formidable task is to find parameters that will achieve low loss on data that we have not seen before, a challenge called *generalization*. We return to these topics throughout the book.

Making Predictions with the Learned Model

Given the learned linear regression model $\hat{\mathbf{w}}^\top x + \hat{b}$, we can now estimate the price of a new house (not contained in the training data) given its area x_1 and age (year) x_2. Estimating targets given features is commonly called *prediction* and *inference*.

We will try to stick with *prediction* because calling this step *inference*, despite emerging as standard jargon in deep learning, is somewhat of a misnomer. In statistics, *inference* more often denotes estimating parameters based on a dataset. This misuse of terminology is a common source of confusion when deep learning practitioners talk to statisticians.

Vectorization for Speed

When training our models, we typically want to process whole minibatches of examples simultaneously. Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.

To illustrate why this matters so much, we can consider two methods for adding vectors. To start we instantiate two 100000-dimensional vectors containing all ones. In one method we will loop over the vectors with a Python `for` loop. In the other method we will rely on a single call to `np`.

```
%matplotlib inline
import d2l
import math
from mxnet import np
import time

n = 10000
a = np.ones(n)
b = np.ones(n)
```

Since we will benchmark the running time frequently in this book, let's define a timer (hereafter accessed via the `d2l` package to track the running time.

```
# Saved in the d2l package for later use
class Timer(object):
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        # Start the timer
        self.start_time = time.time()

    def stop(self):
        # Stop the timer and record the time in a list
        self.times.append(time.time() - self.start_time)
        return self.times[-1]

    def avg(self):
        # Return the average time
        return sum(self.times)/len(self.times)

    def sum(self):
        # Return the sum of time
        return sum(self.times)

    def cumsum(self):
        # Return the accumuated times
        return np.array(self.times).cumsum().tolist()
```

Now we can benchmark the workloads. First, we add them, one coordinate at a time, using a `for` loop.

```
timer = Timer()
c = np.zeros(n)
for i in range(n):
    c[i] = a[i] + b[i]
'%.5f sec' % timer.stop()
```

```
'3.51575 sec'
```

Alternatively, we rely on `np` to compute the elementwise sum:

```
timer.start()
d = a + b
'%.5f sec' % timer.stop()
```

```
'0.00023 sec'
```

You probably noticed that the second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the math to the library and need not write as many calculations ourselves, reducing the potential for errors.

3.1.2 The Normal Distribution and Squared Loss

While you can already get your hands dirty using only the information above, in the following section we can more formally motivate the square loss objective via assumptions about the distribution of noise.

Recall from the above that the squared loss $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ has many convenient properties. These include a simple derivative $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$.

As we mentioned earlier, linear regression was invented by Gauss in 1795, who also discovered the normal distribution (also called the *Gaussian*). It turns out that the connection between the normal distribution and linear regression runs deeper than common parentage. To refresh your memory, the probability density of a normal distribution with mean μ and variance σ^2 is given as follows:

$$p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(z - \mu)^2\right). \tag{3.1.11}$$

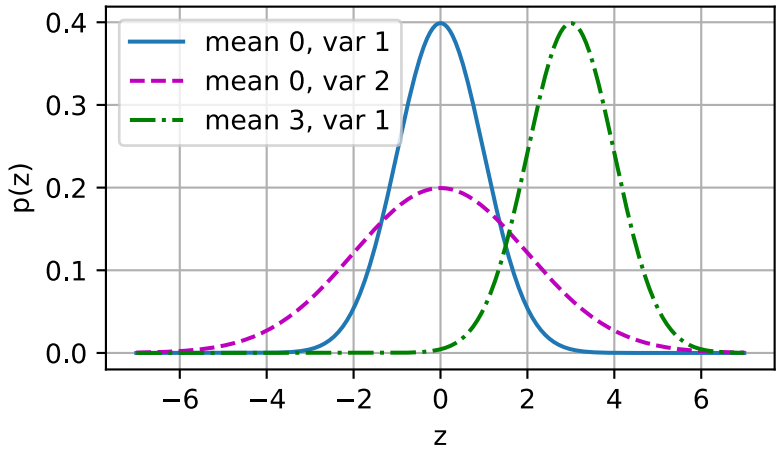
Below we define a Python function to compute the normal distribution.

```
x = np.arange(-7, 7, 0.01)

def normal(z, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(- 0.5 / sigma**2 * (z - mu)**2)
```

We can now visualize the normal distributions.

```
# Mean and variance pairs
parameters = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in parameters], xlabel='z',
         ylabel='p(z)', figsize=(4.5, 2.5),
         legend=['mean %d, var %d' % (mu, sigma) for mu, sigma in parameters])
```



As you can see, changing the mean corresponds to a shift along the *x axis*, and increasing the variance spreads the distribution out, lowering its peak.

One way to motivate linear regression with the mean squared error loss function is to formally assume that observations arise from noisy observations, where the noise is normally distributed as follows

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{3.1.12}$$

Thus, we can now write out the *likelihood* of seeing a particular y for a given $\mathbf{x}$ via

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{3.1.13}$$

Now, according to the *maximum likelihood principle*, the best values of b and $\mathbf{w}$ are those that maximize the *likelihood* of the entire dataset:

$$P(Y \mid X) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}). \tag{3.1.14}$$

Estimators chosen according to the *maximum likelihood principle* are called *Maximum Likelihood Estimators* (MLE). While, maximizing the product of many exponential functions, might look difficult, we can simplify things significantly, without changing the objective, by maximizing the log of the likelihood instead. For historical reasons, optimizations are more often expressed as minimization rather than maximization. So, without changing anything we can minimize the *Negative Log-Likelihood (NLL)* $-\log p(\mathbf{y}|\mathbf{X})$. Working out the math gives us:

$$-\log p(\mathbf{y}|\mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{3.1.15}$$

Now we just need one more assumption: that σ is some fixed constant. Thus we can ignore the first term because it does not depend on $\mathbf{w}$ or b. Now the second term is identical to the squared error objective introduced earlier, but for the multiplicative constant $\frac{1}{\sigma^2}$. Fortunately, the solution does not depend on σ. It follows that minimizing squared error is equivalent to maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

3.1.3 From Linear Regression to Deep Networks

So far we only talked about linear functions. While neural networks cover a much richer family of models, we can begin thinking of the linear model as a neural network by expressing it the language of neural networks. To begin, let's start by rewriting things in a 'layer' notation.

Neural Network Diagram

Deep learning practitioners like to draw diagrams to visualize what is happening in their models. In Fig. 3.1.2, we depict our linear model as a neural network. Note that these diagrams indicate the connectivity pattern (here, each input is connected to the output) but not the values taken by the weights or biases.

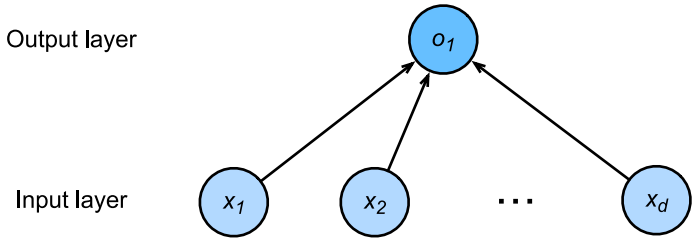


Fig. 3.1.2: Linear regression is a single-layer neural network.

Because there is just a single computed neuron (node) in the graph (the input values are not computed but given), we can think of linear models as neural networks consisting of just a single artificial neuron. Since for this model, every input is connected to every output (in this case there is only one output!), we can regard this transformation as a *fully-connected layer*, also commonly called a *dense layer*. We will talk a lot more about networks composed of such layers in the next chapter on multilayer perceptrons.

Biology

Although linear regression (invented in 1795) predates computational neuroscience, so it might seem anachronistic to describe linear regression as a neural network. To see why linear models were a natural place to begin when the cyberneticists/neurophysiologists Warren McCulloch and Walter Pitts looked when they began to develop models of artificial neurons, consider the cartoonish picture of a biological neuron in Fig. 3.1.3, consisting of *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals), enabling connections to other neurons via *synapses*.

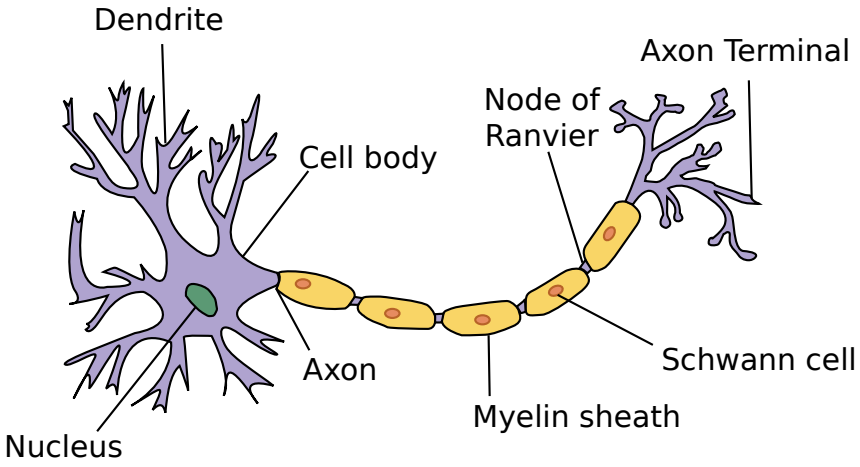


Fig. 3.1.3: The real neuron

Information x_i arriving from other neurons (or environmental sensors such as the retina) is received in the dendrites. In particular, that information is weighted by *synaptic weights* w_i determining the effect of the inputs (e.g., activation or inhibition via the product $x_i w_i$). The weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_i x_i w_i + b$, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via $\sigma(y)$. From there it either reaches its destination (e.g., a muscle) or is fed into another neuron via its dendrites.

Certainly, the high-level idea that many such units could be cobbled together with the right connectivity and right learning algorithm, to produce far more interesting and complex behavior than any one neuron along could express owes to our study of real biological neural systems.

At the same time, most research in deep learning today draws little direct inspiration in neuroscience. We invoke Stuart Russell and Peter Norvig who, in their classic AI text book *Artificial Intelligence: A Modern Approach* (Russell & Norvig, 2016), pointed out that although airplanes might have been *inspired* by birds, ornithology has not been the primary driver of aeronautics innovation for some centuries. Likewise, inspiration in deep learning these days comes in equal or greater measure from mathematics, statistics, and computer science.

Summary

- Key ingredients in a machine learning model are training data, a loss function, an optimization algorithm, and quite obviously, the model itself.
- Vectorizing makes everything better (mostly math) and faster (mostly code).
- Minimizing an objective function and performing maximum likelihood can mean the same thing.
- Linear models are neural networks, too.

Exercises

1. Assume that we have some data $x_1, \ldots, x_n \in \mathbb{R}$. Our goal is to find a constant b such that $\sum_i (x_i - b)^2$ is minimized.
   - Find a closed-form solution for the optimal value of b.
   - How does this problem and its solution relate to the normal distribution?
2. Derive the closed-form solution to the optimization problem for linear regression with squared error. To keep things simple, you can omit the bias b from the problem (we can do this in principled fashion by adding one column to X consisting of all ones).
   - Write out the optimization problem in matrix and vector notation (treat all the data as a single matrix, all the target values as a single vector).
   - Compute the gradient of the loss with respect to w.
   - Find the closed form solution by setting the gradient equal to zero and solving the matrix equation.
   - When might this be better than using stochastic gradient descent? When might this method break?
3. Assume that the noise model governing the additive noise ϵ is the exponential distribution. That is, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
   - Write out the negative log-likelihood of the data under the model $-\log P(Y \mid X)$.
   - Can you find a closed form solution?
   - Suggest a stochastic gradient descent algorithm to solve this problem. What could possibly go wrong (hint - what happens near the stationary point as we keep on updating the parameters). Can you fix this?

3.2 Linear Regression Implementation from Scratch

Now that you understand the key ideas behind linear regression, we can begin to work through a hands-on implementation in code. In this section, we will implement the entire method from scratch, including the data pipeline, the model, the loss function, and the gradient descent optimizer. While modern deep learning frameworks can automate nearly all of this work, implementing things from scratch is the only to make sure that you really know what you are doing. Moreover, when it comes time to customize models, defining our own layers, loss functions, etc., understanding how things work under the hood will prove handy. In this section, we will rely only on `ndarray` and `autograd`. Afterwards, we will introduce a more compact implementation, taking advantage of Gluon's bells and whistles. To start off, we import the few required packages.

```
%matplotlib inline
import d2l
```

(continues on next page)

(continued from previous page)

```
from mxnet import autograd, np, npx
import random
npx.set_np()
```

3.2.1 Generating the Dataset

To keep things simple, we will construct an artificial dataset according to a linear model with additive noise. Out task will be to recover this model's parameters using the finite set of examples contained in our dataset. We will keep the data low-dimensional so we can visualize it easily. In the following code snippet, we generated a dataset containing 1000 examples, each consisting of 2 features sampled from a standard normal distribution. Thus our synthetic dataset will be an object $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

The true parameters generating our data will be $\mathbf{w} = [2, -3.4]^\top$ and $b = 4.2$ and our synthetic labels will be assigned according to the following linear model with noise term ϵ:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \tag{3.2.1}$$

You could think of ϵ as capturing potential measurement errors on the features and labels. We will assume that the standard assumptions hold and thus that ϵ obeys a normal distribution with mean of 0. To make our problem easy, we will set its standard deviation to 0.01. The following code generates our synthetic dataset:

```
# Saved in the d2l package for later use
def synthetic_data(w, b, num_examples):
    """generate y = X w + b + noise"""
    X = np.random.normal(0, 1, (num_examples, len(w)))
    y = np.dot(X, w) + b
    y += np.random.normal(0, 0.01, y.shape)
    return X, y

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

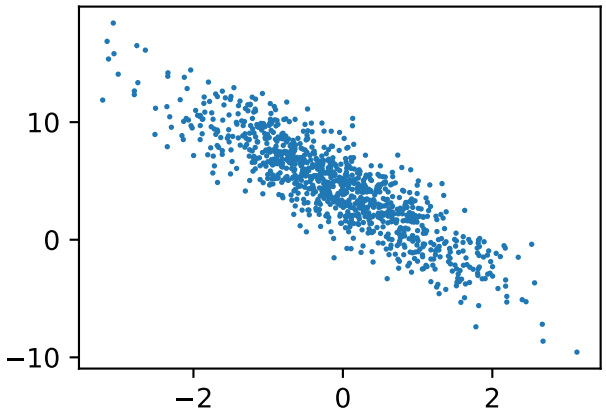
Note that each row in `features` consists of a 2-dimensional data point and that each row in `labels` consists of a 1-dimensional target value (a scalar).

```
print('features:', features[0],'\nlabel:', labels[0])
```

```
features: [2.2122064 1.1630787]
label: 4.662078
```

By generating a scatter plot using the second `features[:, 1]` and `labels`, we can clearly observe the linear correlation between the two.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```



3.2.2 Reading the Dataset

Recall that training models consists of making multiple passes over the dataset, grabbing one minibatch of examples at a time, and using them to update our model. Since this process is so fundamental to training machine learning algorithms, its worth defining a utility function to shuffle the data and access it in minibatches.

In the following code, we define a `data_iter` function to demonstrate one possible implementation of this functionality. The function takes a batch size, a design matrix, and a vector of labels, yielding minibatches of size `batch_size`. Each minibatch consists of an tuple of features and labels.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = np.array(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

In general, note that we want to use reasonably sized minibatches to take advantage of the GPU hardware, which excels at parallelizing operations. Because each example can be fed through our models in parallel and the gradient of the loss function for each example can also be taken in parallel, GPUs allow us to process hundreds of examples in scarcely more time than it might take to process just a single example.

To build some intuition, let's read and print the first small batch of data examples. The shape of the features in each minibatch tells us both the minibatch size and the number of input features. Likewise, our minibatch of labels will have a shape given by `batch_size`.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
[[ 1.0779219   0.38224545]
 [ 0.24021588 -0.5396039 ]
 [-0.66915834 -1.197868  ]
 [ 0.11402581 -1.1407781 ]
 [ 1.015367    1.2454321 ]
 [ 0.71535987  1.8271433 ]
 [ 1.0240983   0.56759083]
 [ 1.1291474  -0.41478267]
 [-1.6327407   0.07702067]
 [ 0.3925466  -0.18325637]]
 [ 5.03691     6.494923    6.9462047   8.31074     2.0036945  -0.57446426
  4.3213058   7.877404    0.6821707   5.610293  ]
```

As we run the iterator, we obtain distinct minibatches successively until all the data has been exhausted (try this). While the iterator implemented above is good for didactic purposes, it is inefficient in ways that might get us in trouble on real problems. For example, it requires that we load all data in memory and that we perform lots of random memory access. The built-in iterators implemented in Apache MXNet are considerably efficient and they can deal both with data stored on file and data fed via a data stream.

3.2.3 Initializing Model Parameters

Before we can begin optimizing our model's parameters by gradient descent, we need to have some parameters in the first place. In the following code, we initialize weights by sampling random numbers from a normal distribution with mean 0 and a standard deviation of 0.01, setting the bias b to 0.

```
w = np.random.normal(0, 0.01, (2, 1))
b = np.zeros(1)
```

Now that we have initialized our parameters, our next task is to update them until they fit our data sufficiently well. Each update requires taking the gradient (a multi-dimensional derivative) of our loss function with respect to the parameters. Given this gradient, we can update each parameter in the direction that reduces the loss.

Since nobody wants to compute gradients explicitly (this is tedious and error prone), we use automatic differentiation to compute the gradient. See Section 2.5 for more details. Recall from the autograd chapter that in order for `autograd` to know that it should store a gradient for our parameters, we need to invoke the `attach_grad` function, allocating memory to store the gradients that we plan to take.

```
w.attach_grad()
b.attach_grad()
```

3.2.4 Defining the Model

Next, we must define our model, relating its inputs and parameters to its outputs. Recall that to calculate the output of the linear model, we simply take the matrix-vector dot product of the examples $\mathbf{X}$ and the models weights w, and add the offset b to each example. Note that below `np.dot(X, w)` is a vector and `b` is a scalar. Recall that when we add a vector and a scalar, the scalar is added to each component of the vector.

```
# Saved in the d2l package for later use
def linreg(X, w, b):
    return np.dot(X, w) + b
```

3.2.5 Defining the Loss Function

Since updating our model requires taking the gradient of our loss function, we ought to define the loss function first. Here we will use the squared loss function as described in the previous section. In the implementation, we need to transform the true value `y` into the predicted value's shape `y_hat`. The result returned by the following function will also be the same as the `y_hat` shape.

```
# Saved in the d2l package for later use
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

3.2.6 Defining the Optimization Algorithm

As we discussed in the previous section, linear regression has a closed-form solution. However, this is not a book about linear regression, it is a book about deep learning. Since none of the other models that this book introduces can be solved analytically, we will take this opportunity to introduce your first working example of stochastic gradient descent (SGD).

At each step, using one batch randomly drawn from our dataset, we will estimate the gradient of the loss with respect to our parameters. Next, we will update our parameters (a small amount) in the direction that reduces the loss. Recall from Section 2.5 that after we call `backward` each parameter (`param`) will have its gradient stored in `param.grad`. The following code applies the SGD update, given a set of parameters, a learning rate, and a batch size. The size of the update step is determined by the learning rate `lr`. Because our loss is calculated as a sum over the batch of examples, we normalize our step size by the batch size (`batch_size`), so that the magnitude of a typical step size does not depend heavily on our choice of the batch size.

```
# Saved in the d2l package for later use
def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

3.2.7 Training

Now that we have all of the parts in place, we are ready to implement the main training loop. It is crucial that you understand this code because you will see nearly identical training loops over and over again throughout your career in deep learning.

In each iteration, we will grab minibatches of models, first passing them through our model to obtain a set of predictions. After calculating the loss, we call the `backward` function to initiate the backwards pass through the network, storing the gradients with respect to each parameter in its corresponding `.grad` attribute. Finally, we will call the optimization algorithm `sgd` to update the model parameters. Since we previously set the batch size `batch_size` to 10, the loss shape `l` for each minibatch is $(10, 1)$.

In summary, we will execute the following loop:

- Initialize parameters $(\mathbf{w}, b)$
- Repeat until done
  - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} l(\mathbf{x}^i, y^i, \mathbf{w}, b)$
  - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

In the code below, `l` is a vector of the losses for each example in the minibatch. Because `l` is not a scalar variable, running `l.backward()` adds together the elements in `l` to obtain the new variable and then calculates the gradient.

In each epoch (a pass through the data), we will iterate through the entire dataset (using the `data_iter` function) once passing through every examples in the training dataset (assuming the number of examples is divisible by the batch size). The number of epochs `num_epochs` and the learning rate `lr` are both hyper-parameters, which we set here to 3 and 0.03, respectively. Unfortunately, setting hyper-parameters is tricky and requires some adjustment by trial and error. We elide these details for now but revise them later in Chapter 11.

```
lr = 0.03  # Learning rate
num_epochs = 3  # Number of iterations
net = linreg  # Our fancy linear model
loss = squared_loss  # 0.5 (y-y')^2

for epoch in range(num_epochs):
    # Assuming the number of examples can be divided by the batch size, all
    # the examples in the training dataset are used once in one epoch
    # iteration. The features and tags of minibatch examples are given by X
    # and y respectively
    for X, y in data_iter(batch_size, features, labels):
        with autograd.record():
            l = loss(net(X, w, b), y)  # Minibatch loss in X and y
        l.backward()  # Compute gradient on l with respect to [w, b]
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    train_l = loss(net(features, w, b), labels)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))
```

```
epoch 1, loss 0.024923
epoch 2, loss 0.000091
epoch 3, loss 0.000051
```

In this case, because we synthesized the data ourselves, we know precisely what the true parameters are. Thus, we can evaluate our success in training by comparing the true parameters with those that we learned through our training loop. Indeed they turn out to be very close to each other.

```
print('Error in estimating w', true_w - w.reshape(true_w.shape))
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [ 5.5313110e-05 -2.6226044e-06]
Error in estimating b [0.00063753]
```

Note that we should not take it for granted that we are able to recover the parameters accurately. This only happens for a special category problems: strongly convex optimization problems with "enough" data to ensure that the noisy samples allow us to recover the underlying dependency. In most cases this is *not* the case. In fact, the parameters of a deep network are rarely the same (or even close) between two different runs, unless all conditions are identical, including the order in which the data is traversed. However, in machine learning, we are typically less concerned with recovering true underlying parameters, and more concerned with parameters that lead to accurate prediction. Fortunately, even on difficult optimization problems, stochastic gradient descent can often find remarkably good solutions, owing partly to the fact that, for deep networks, there exist many configurations of the parameters that lead to accurate prediction.

Summary

We saw how a deep network can be implemented and optimized from scratch, using just `ndarray` and `autograd`, without any need for defining layers, fancy optimizers, etc. This only scratches the surface of what is possible. In the following sections, we will describe additional models based on the concepts that we have just introduced and learn how to implement them more concisely.

Exercises

1. What would happen if we were to initialize the weights $\mathbf{w} = 0$. Would the algorithm still work?
2. Assume that you are Georg Simon Ohm[51] trying to come up with a model between voltage and current. Can you use `autograd` to learn the parameters of your model.
3. Can you use Planck's Law[52] to determine the temperature of an object using spectral energy density?
4. What are the problems you might encounter if you wanted to extend `autograd` to second derivatives? How would you fix them?
5. Why is the `reshape` function needed in the `squared_loss` function?
6. Experiment using different learning rates to find out how fast the loss function value drops.
7. If the number of examples cannot be divided by the batch size, what happens to the `data_iter` function's behavior?

[51] https://en.wikipedia.org/wiki/Georg_Ohm
[52] https://en.wikipedia.org/wiki/Planck%27s_law

3.3 Concise Implementation of Linear Regression

Broad and intense interest in deep learning for the past several years has inspired both companies, academics, and hobbyists to develop a variety of mature open source frameworks for automating the repetitive work of implementing gradient-based learning algorithms. In the previous section, we relied only on (i) `ndarray` for data storage and linear algebra; and (ii) `autograd` for calculating derivatives. In practice, because data iterators, loss functions, optimizers, and neural network layers (and some whole architectures) are so common, modern libraries implement these components for us as well.

In this section, we will show you how to implement the linear regression model from Section 3.2 concisely by using Gluon.

3.3.1 Generating the Dataset

To start, we will generate the same dataset as in the previous section.

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

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

3.3.2 Reading the Dataset

Rather than rolling our own iterator, we can call upon Gluon's `data` module to read data. The first step will be to instantiate an `ArrayDataset`. This object's constructor takes one or more `ndarrays` as arguments. Here, we pass in `features` and `labels` as arguments. Next, we will use the `ArrayDataset` to instantiate a `DataLoader`, which also requires that we specify a `batch_size` and specify a Boolean value `shuffle` indicating whether or not we want the `DataLoader` to shuffle the data on each epoch (pass through the dataset).

```
# Saved in the d2l package for later use
def load_array(data_arrays, batch_size, is_train=True):
    """Construct a Gluon data loader"""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Now we can use `data_iter` in much the same way as we called the `data_iter` function in the previous section. To verify that it is working, we can read and print the first minibatch of instances.

```
for X, y in data_iter:
    print(X, '\n', y)
    break
```

```
[[ 1.9118724   0.99435186]
 [-0.69279176 -0.11865307]
 [-0.83655155 -0.58954424]
 [-0.9345107   1.0642178 ]
 [-0.34534106 -0.7556803 ]
 [ 0.30967948 -1.7559303 ]
 [-0.6308071   0.53733146]
 [ 0.46879596 -0.30394194]
 [ 0.75600237  0.16707687]
 [-0.16452314 -0.21437684]]
 [ 4.6377006  3.2253058  4.530804  -1.2874792  6.07641   10.791387
  1.0948168  6.1717677  5.1450267  4.620073 ]
```

3.3.3 Defining the Model

When we implemented linear regression from scratch (in :numref``sec_linear_scratch``), we defined our model parameters explicitly and coded up the calculations to produce output using basic linear algebra operations. You *should* know how to do this. But once your models get more complex, and once you have to do this nearly every day, you will be glad for the assistance. The situation is similar to coding up your own blog from scratch. Doing it once or twice is rewarding and instructive, but you would be a lousy web developer if every time you needed a blog you spent a month reinventing the wheel.

For standard operations, we can use Gluon's predefined layers, which allow us to focus especially on the layers used to construct the model rather than having to focus on the implementation. To define a linear model, we first import the `nn` module, which defines a large number of neural network layers (note that "nn" is an abbreviation for neural networks). We will first define a model variable `net`, which will refer to an instance of the `Sequential` class. In Gluon, `Sequential` defines a container for several layers that will be chained together. Given input data, a `Sequential` passes it through the first layer, in turn passing the output as the second layer's input and so forth. In the following example, our model consists of only one layer, so we do not really need `Sequential`. But since nearly all of our future models will involve multiple layers, we will use it anyway just to familiarize you with the most standard workflow.

```
from mxnet.gluon import nn
net = nn.Sequential()
```

Recall the architecture of a single-layer network as shown in Fig. 3.3.1. The layer is said to be *fully-connected* because each of its inputs are connected to each of its outputs by means of a matrix-vector multiplication. In Gluon, the fully-connected layer is defined in the `Dense` class. Since we only want to generate a single scalar output, we set that number to 1.

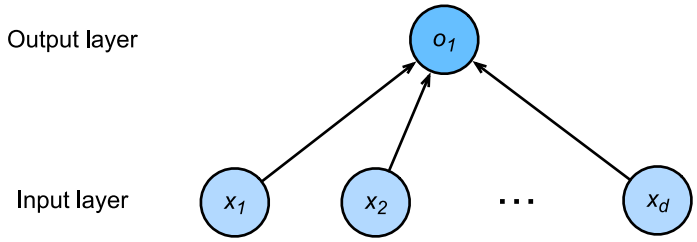


Fig. 3.3.1: Linear regression is a single-layer neural network.

```
net.add(nn.Dense(1))
```

It is worth noting that, for convenience, Gluon does not require us to specify the input shape for each layer. So here, we do not need to tell Gluon how many inputs go into this linear layer. When we first try to pass data through our model, e.g., when we execute `net(X)` later, Gluon will automatically infer the number of inputs to each layer. We will describe how this works in more detail in the chapter "Deep Learning Computation".

3.3.4 Initializing Model Parameters

Before using `net`, we need to initialize the model parameters, such as the weights and biases in the linear regression model. We will import the `initializer` module from MXNet. This module provides various methods for model parameter initialization. Gluon makes `init` available as a shortcut (abbreviation) to access the `initializer` package. By calling `init.Normal(sigma=0.01)`, we specify that each *weight* parameter should be randomly sampled from a normal distribution with mean 0 and standard deviation 0.01. The *bias* parameter will be initialized to zero by default. Both the weight vector and bias will have attached gradients.

```
from mxnet import init
net.initialize(init.Normal(sigma=0.01))
```

The code above may look straightforward but you should note that something strange is happening here. We are initializing parameters for a network even though Gluon does not yet know how many dimensions the input will have! It might be 2 as in our example or it might be 2000. Gluon lets us get away with this because behind the scenes, the initialization is actually *deferred*. The real initialization will take place only when we for the first time attempt to pass data through the network. Just be careful to remember that since the parameters have not been initialized yet, we cannot access or manipulate them.

3.3.5 Defining the Loss Function

In Gluon, the `loss` module defines various loss functions. We will the imported module `loss` with the pseudonym `gloss`, to avoid confusing it for the variable holding our chosen loss function. In this example, we will use the Gluon implementation of squared loss (`L2Loss`).

```
from mxnet.gluon import loss as gloss
loss = gloss.L2Loss()  # The squared loss is also known as the L2 norm loss
```

3.3.6 Defining the Optimization Algorithm

Minibatch SGD and related variants are standard tools for optimizing neural networks and thus Gluon supports SGD alongside a number of variations on this algorithm through its `Trainer` class. When we instantiate the `Trainer`, we will specify the parameters to optimize over (obtainable from our net via `net.collect_params()`), the optimization algorithm we wish to use (`sgd`), and a dictionary of hyper-parameters required by our optimization algorithm. SGD just requires that we set the value `learning_rate`, (here we set it to 0.03).

```
from mxnet import gluon
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

3.3.7 Training

You might have noticed that expressing our model through Gluon requires comparatively few lines of code. We did not have to individually allocate parameters, define our loss function, or implement stochastic gradient descent. Once we start working with much more complex models, Gluon's advantages will grow considerably. However, once we have all the basic pieces in place, the training loop itself is strikingly similar to what we did when implementing everything from scratch.

To refresh your memory: for some number of epochs, we will make a complete pass over the dataset (train_data), iteratively grabbing one minibatch of inputs and the corresponding ground-truth labels. For each minibatch, we go through the following ritual:

- Generate predictions by calling `net(X)` and calculate the loss `l` (the forward pass).
- Calculate gradients by calling `l.backward()` (the backward pass).
- Update the model parameters by invoking our SGD optimizer (note that `trainer` already knows which parameters to optimize over, so we just need to pass in the minibatch size.

For good measure, we compute the loss after each epoch and print it to monitor progress.

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
        l.backward()
        trainer.step(batch_size)
    l = loss(net(features), labels)
    print('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))
```

```
epoch 1, loss: 0.025167
epoch 2, loss: 0.000090
epoch 3, loss: 0.000051
```

Below, we compare the model parameters learned by training on finite data and the actual parameters that generated our dataset. To access parameters with Gluon, we first access the layer that we need from `net` and then access that layer's weight (`weight`) and bias (`bias`). To access each parameter's values as an `ndarray`, we invoke its `data` method. As in our from-scratch implementation, note that our estimated parameters are close to their ground truth counterparts.

```
w = net[0].weight.data()
print('Error in estimating w', true_w.reshape(w.shape) - w)
b = net[0].bias.data()
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [[0.00019813 0.0003016 ]]
Error in estimating b [0.00057745]
```

Summary

- Using Gluon, we can implement models much more succinctly.
- In Gluon, the `data` module provides tools for data processing, the `nn` module defines a large number of neural network layers, and the `loss` module defines many common loss functions.
- MXNet's module `initializer` provides various methods for model parameter initialization.
- Dimensionality and storage are automatically inferred (but be careful not to attempt to access parameters before they have been initialized).

Exercises

1. If we replace `l = loss(output, y)` with `l = loss(output, y).mean()`, we need to change `trainer.step(batch_size)` to `trainer.step(1)` for the code to behave identically. Why?
2. Review the MXNet documentation to see what loss functions and initialization methods are provided in the modules `gluon.loss` and `init`. Replace the loss by Huber's loss.
3. How do you access the gradient of `dense.weight`?

3.4 Softmax Regression

In Section 3.1, we introduced linear regression, working through implementations from scratch in Section 3.2 and again using Gluon in Section 3.3 to do the heavy lifting.

Regression is the hammer we reach for when we want to answer *how much?* or *how many?* questions. If you want to predict the number of dollars (the *price*) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you are probably looking for a regression model.

In practice, we are more often interested in classification: asking not *how much?* but *which one?*

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

- Does this image depict a donkey, a dog, a cat, or a rooster?
- Which movie is Aston most likely to watch next?

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

3.4.1 Classification Problems

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

Next, we have to choose how to represent the labels. We have two obvious choices. Perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent {dog, cat, chicken} respectively. This is a great way of *storing* such information on a computer. If the categories had some natural ordering among them, say if we were trying to predict {baby, toddler, adolescent, young adult, adult, geriatric}, then it might even make sense to cast this problem as regression and keep the labels in this format.

But general classification problems do not come with natural orderings among the classes. Fortunately, statisticians long ago invented a simple way to represent categorical data: the *one hot encoding*. A one-hot encoding is a vector with as many components as we have categories. The component corresponding to particular instance's category is set to 1 and all other components are set to 0.

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{3.4.1}$$

In our case, y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken" and $(0, 0, 1)$ to "dog".

Network Architecture

In order to estimate the conditional probabilities associated with each classes, we need a model with multiple outputs, one per class. To address classification with linear models, we will need as many linear functions as we have outputs. Each output will correspond to its own linear function. In our case, since we have 4 features and 3 possible output categories, we will need 12 scalars to represent the weights, (w with subscripts) and 3 scalars to represent the biases (b with subscripts). We compute these three *logits*, o_1, o_2, and o_3, for each input:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{3.4.2}$$

We can depict this calculation with the neural network diagram shown in Fig. 3.4.1. Just as in linear regression, softmax regression is also a single-layer neural network. And since the calculation of each output, o_1, o_2, and o_3, depends on all inputs, x_1, x_2, x_3, and x_4, the output layer of softmax regression can also be described as fully-connected layer.

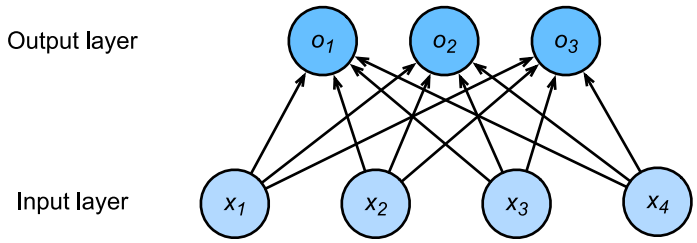


Fig. 3.4.1: Softmax regression is a single-layer neural network.

To express the model more compactly, we can use linear algebra notation. In vector form, we arrive at $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, a form better suited both for mathematics, and for writing code. Note that we have gathered all of our weights into a 3×4 matrix and that for a given example $\mathbf{x}$, our outputs are given by a matrix-vector product of our weights by our inputs plus our biases $\mathbf{b}$.

Softmax Operation

The main approach that we are going to take here is to interpret the outputs of our model as probabilities. We will optimize our parameters to produce probabilities that maximize the likelihood of the observed data. Then, to generate predictions, we will set a threshold, for example, choosing the *argmax* of the predicted probabilities.

Put formally, we would like outputs $\hat{y}_k$ that we can interpret as the probability that a given item belongs to class k. Then we can choose the class with the largest output value as our prediction $\operatorname{argmax}_k y_k$. For example, if $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$ are 0.1, $.8$, and 0.1, respectively, then we predict category 2, which (in our example) represents "chicken".

You might be tempted to suggest that we interpret the logits o directly as our outputs of interest. However, there are some problems with directly interpreting the output of the linear layer as a probability. Nothing constrains these numbers to sum to 1. Moreover, depending on the inputs, they can take negative values. These violate basic axioms of probability presented in Section 2.6

To interpret our outputs as probabilities, we must guarantee that (even on new data), they will be nonnegative and sum up to 1. Moreover, we need a training objective that encourages the model to estimate faithfully *probabilities*. Of all instances when a classifier outputs $.5$, we hope that half of those examples will *actually* belong to the predicted class. This is a property called *calibration*.

The *softmax function*, invented in 1959 by the social scientist R Duncan Luce in the context of *choice models* does precisely this. To transform our logits such that they become nonnegative and sum to 1, while requiring that the model remains differentiable, we first exponentiate each logit (ensuring non-negativity) and then divide by their sum (ensuring that they sum to 1).

$$\hat{\mathbf{y}} = \operatorname{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}. \tag{3.4.3}$$

It is easy to see $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ with $0 \leq \hat{y}_i \leq 1$ for all i. Thus, $\hat{y}$ is a proper probability distribution and the values of $\hat{\mathbf{y}}$ can be interpreted accordingly. Note that the softmax operation does not change the ordering among the logits, and thus we can still pick out the most likely class by:

$$\hat{\imath}(\mathbf{o}) = \operatorname*{argmax}_i o_i = \operatorname*{argmax}_i \hat{y}_i. \tag{3.4.4}$$

The logits $\mathbf{o}$ then are simply the pre-softmax values that determining the probabilities assigned to each category. Summarizing it all in vector notation we get $\mathbf{o}^{(i)} = \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}$, where $\hat{\mathbf{y}}^{(i)} = \operatorname{softmax}(\mathbf{o}^{(i)})$.

Vectorization for Minibatches

To improve computational efficiency and take advantage of GPUs, we typically carry out vector calculations for minibatches of data. Assume that we are given a minibatch $\mathbf{X}$ of examples with dimensionality d and batch size n. Moreover, assume that we have q categories (outputs). Then the minibatch features $\mathbf{X}$ are in $\mathbb{R}^{n \times d}$, weights $\mathbf{W} \in \mathbb{R}^{d \times q}$, and the bias satisfies $\mathbf{b} \in \mathbb{R}^q$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned} \tag{3.4.5}$$

This accelerates the dominant operation into a matrix-matrix product $\mathbf{W}\mathbf{X}$ vs the matrix-vector products we would be executing if we processed one example at a time. The softmax itself can be computed by exponentiating all entries in $\mathbf{O}$ and then normalizing them by the sum.

3.4.2 Loss Function

Next, we need a *loss function* to measure the quality of our predicted probabilities. We will rely on *likelihood maximization*, the very same concept that we encountered when providing a probabilistic justification for the least squares objective in linear regression (Section 3.1).

Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as estimated conditional probabilities of each class given the input x, e.g., $\hat{y}_1 = \hat{P}(y = \text{cat} \mid \mathbf{x})$. We can compare the estimates with reality by checking how probable the *actual* classes are according to our model, given the features.

$$P(Y \mid X) = \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}) \text{ and thus } -\log P(Y \mid X) = \sum_{i=1}^{n} -\log P(y^{(i)} \mid x^{(i)}). \tag{3.4.6}$$

Maximizing $P(Y \mid X)$ (and thus equivalently minimizing $-\log P(Y \mid X)$) corresponds to predicting the label well. This yields the loss function (we dropped the superscript (i) to avoid notation clutter):

$$l = -\log P(y \mid x) = -\sum_{j} y_j \log \hat{y}_j. \tag{3.4.7}$$

For reasons explained later on, this loss function is commonly called the *cross-entropy* loss. Here, we used that by construction $\hat{y}$ is a discrete probability distribution and that the vector $\mathbf{y}$ is a one-hot vector. Hence the the sum over all coordinates j vanishes for all but one term. Since all $\hat{y}_j$ are probabilities, their logarithm is never larger than 0. Consequently, the loss function cannot be minimized any further if we correctly predict y with *certainty*, i.e., if $P(y \mid x) = 1$ for the correct label. Note that this is often not possible. For example, there might be label noise in the dataset (some examples may be mislabeled). It may also not be possible when the input features are not sufficiently informative to classify every example perfectly.

Softmax and Derivatives

Since the softmax and the corresponding loss are so common, it is worth while understanding a bit better how it is computed. Plugging o into the definition of the loss l and using the definition of the softmax we obtain:

$$l = -\sum_j y_j \log \hat{y}_j = \sum_j y_j \log \sum_k \exp(o_k) - \sum_j y_j o_j = \log \sum_k \exp(o_k) - \sum_j y_j o_j. \tag{3.4.8}$$

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

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j = P(y = j \mid x) - y_j. \tag{3.4.9}$$

In other words, the gradient is the difference between the probability assigned to the true class by our model, as expressed by the probability $P(y \mid x)$, and what actually happened, as expressed by y. In this sense, it is very similar to what we saw in regression, where the gradient was the difference between the observation y and estimate $\hat{y}$. This is not coincidence. In any exponential family[55] model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Cross-Entropy Loss

Now consider the case where we observe not just a single outcome but an entire distribution over outcomes. We can use the same representation as before for y. The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l still works out fine, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_j y_j \log \hat{y}_j. \tag{3.4.10}$$

This loss is called the cross-entropy loss and it is one of the most commonly used losses for multi-class classification. We can demystify the name by introducing the basics of information theory.

3.4.3 Information Theory Basics

Information theory deals with the problem of encoding, decoding, transmitting and manipulating information (also known as data) in as concise form as possible.

Entropy

The central idea in information theory is to quantify the information content in data. This quantity places a hard limit on our ability to compress the data. In information theory, this quantity is called the entropy[56] of a distribution p, and it is captured by the following equation:

$$H[p] = \sum_j -p(j) \log p(j). \tag{3.4.11}$$

[55] https://en.wikipedia.org/wiki/Exponential_family
[56] https://en.wikipedia.org/wiki/Entropy

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

Surprisal

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

However if we cannot perfectly predict every event, then we might some times be surprised. Our surprise is greater when we assigned an event lower probability. For reasons that we will elaborate in the appendix, Claude Shannon settled on $\log(1/p(j)) = -\log p(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $p(j)$. The entropy is then the *expected surprisal* when one assigned the correct probabilities (that truly match the data-generating process). The entropy of the data is then the least surprised that one can ever be (in expectation).

Cross-Entropy Revisited

So if entropy is level of surprise experienced by someone who knows the true probability, then you might be wondering, *what is cross-entropy?* The cross-entropy *from :math:`p` to :math:`q`*, denoted H(p, q), is the expected surprisal of an observer with subjective probabilities q upon seeing data that was actually generated according to probabilities p. The lowest possible cross-entropy is achieved when $p = q$. In this case, the cross-entropy from p to q is $H(p, p) = H(p)$. Relating this back to our classification objective, even if we get the best possible predictions, if the best possible possible, then we will never be perfect. Our loss is lower-bounded by the entropy given by the actual conditional distributions $P(\mathbf{y} \mid \mathbf{x})$.

Kullback Leibler Divergence

Perhaps the most common way to measure the distance between two distributions is to calculate the *Kullback Leibler divergence* $D(p\|q)$. This is simply the difference between the cross-entropy and the entropy, i.e., the additional cross-entropy incurred over the irreducible minimum value it could take:

$$D(p\|q) = H(p, q) - H[p] = \sum_j p(j) \log \frac{p(j)}{q(j)}. \tag{3.4.12}$$

Note that in classification, we do not know the true p, so we cannot compute the entropy directly. However, because the entropy is out of our control, minimizing $D(p\|q)$ with respect to q is equivalent to minimizing the cross-entropy loss.

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

3.4.4 Model Prediction and Evaluation

After training the softmax regression model, given any example features, we can predict the probability of each output category. Normally, we use the category with the highest predicted probability as the output category. The prediction is correct if it is consistent with the actual category (label). In the next part of the experiment, we will use accuracy to evaluate the model's performance. This is equal to the ratio between the number of correct predictions a nd the total number of predictions.

Summary

- We introduced the softmax operation which takes a vector maps it into probabilities.
- Softmax regression applies to classification problems. It uses the probability distribution of the output category in the softmax operation.
- cross-entropy is a good measure of the difference between two probability distributions. It measures the number of bits needed to encode the data given our model.

Exercises

1. Show that the Kullback-Leibler divergence $D(p\|q)$ is nonnegative for all distributions p and q. Hint: use Jensen's inequality, i.e., use the fact that $-\log x$ is a convex function.
2. Show that $\log \sum_j \exp(o_j)$ is a convex function in o.
3. We can explore the connection between exponential families and the softmax in some more depth
   - Compute the second derivative of the cross-entropy loss $l(y, \hat{y})$ for the softmax.
   - Compute the variance of the distribution given by $\text{softmax}(o)$ and show that it matches the second derivative computed above.
4. Assume that we three classes which occur with equal probability, i.e., the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
   - What is the problem if we try to design a binary code for it? Can we match the entropy lower bound on the number of bits?
   - Can you design a better code. Hint: what happens if we try to encode two independent observations? What if we encode n observations jointly?
5. Softmax is a misnomer for the mapping introduced above (but everyone in deep learning uses it). The real softmax is defined as $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
   - Prove that $\text{RealSoftMax}(a, b) > \max(a, b)$.
   - Prove that this holds for $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.
   - Show that for $\lambda \to \infty$ we have $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.
   - What does the soft-min look like?
   - Extend this to more than two numbers.

3.5 The Image Classification Dataset (Fashion-MNIST)

In Section 17.8, we trained a naive Bayes classifier, using the MNIST dataset introduced in 1998 (LeCun et al., 1998). While MNIST had a good run as a benchmark dataset, even simple models by today's standards achieve classification accuracy over 95%. making it unsuitable for distinguishing between stronger models and weaker ones. Today, MNIST serves as more of sanity checks than as a benchmark. To up the ante just a bit, we will focus our discussion in the coming sections on the qualitatively similar, but comparatively complex Fashion-MNIST dataset (Xiao et al., 2017), which was released in 2017.

```
%matplotlib inline
import d2l
from mxnet import gluon
import sys

d2l.use_svg_display()
```

3.5.1 Getting the Dataset

Just as with MNIST, Gluon makes it easy to download and load the FashionMNIST dataset into memory via the `FashionMNIST` class contained in `gluon.data.vision`. We briefly work through the mechanics of loading and exploring the dataset below. Please refer to Section 17.8 for more details on loading data.

```
mnist_train = gluon.data.vision.FashionMNIST(train=True)
mnist_test = gluon.data.vision.FashionMNIST(train=False)
```

FashionMNIST consists of images from 10 categories, each represented by 6k images in the training set and by 1k in the test set. Consequently the training set and the test set contain 60k and 10k images, respectively.

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

The images in Fashion-MNIST are associated with the following categories: t-shirt, trousers, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. The following function converts between numeric label indices and their names in text.

```
# Saved in the d2l package for later use
def get_fashion_mnist_labels(labels):
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
```

(continues on next page)

(continued from previous page)

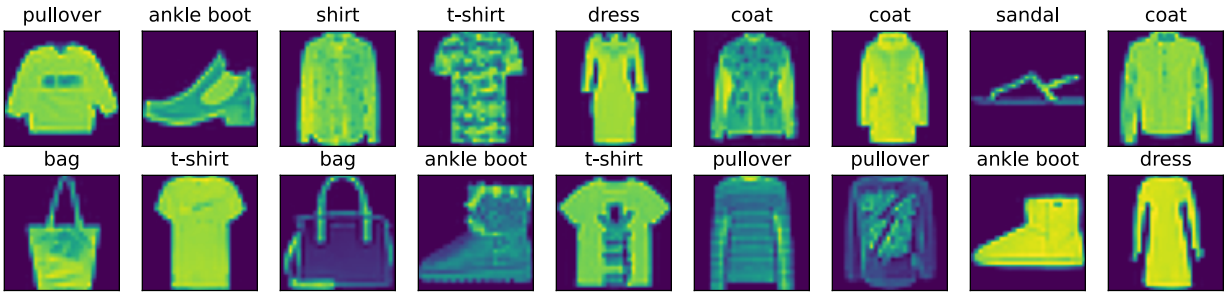
```
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

We can now create a function to visualize these examples.

```
# Saved in the d2l package for later use
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(img.asnumpy())
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

Here are the images and their corresponding labels (in text) for the first few examples in the training dataset.

```
X, y = mnist_train[:18]
show_images(X.squeeze(axis=-1), 2, 9, titles=get_fashion_mnist_labels(y));
```



3.5.2 Reading a Minibatch

To make our life easier when reading from the training and test sets, we use a `DataLoader` rather than creating one from scratch, as we did in Section 3.2. Recall that at each iteration, a `DataLoader` reads a minibatch of data with size `batch_size` each time.

During training, reading data can be a significant performance bottleneck, especially when our model is simple or when our computer is fast. A handy feature of Gluon's `DataLoader` is the ability to use multiple processes to speed up data reading. For instance, we can set aside 4 processes to read the data (via `num_workers`). Because this feature is not currently supported on Windows the following code checks the platform to make sure that we do not saddle our Windows-using friends with error messages later on.

```
# Saved in the d2l package for later use
def get_dataloader_workers(num_workers=4):
```

(continues on next page)

(continued from previous page)

```
    # 0 means no additional process is used to speed up the reading of data.
    if sys.platform.startswith('win'):
        return 0
    else:
        return num_workers
```

Below, we convert the image data from uint8 to 32-bit floating point numbers using the `ToTensor` class. Additionally, the transformer will divide all numbers by 255 so that all pixels have values between 0 and 1. The `ToTensor` class also moves the image channel from the last dimension to the first dimension to facilitate the convolutional neural network calculations introduced later. Through the `transform_first` function of the dataset, we apply the transformation of `ToTensor` to the first element of each instance (image and label).

```
batch_size = 256
transformer = gluon.data.vision.transforms.ToTensor()
train_iter = gluon.data.DataLoader(mnist_train.transform_first(transformer),
                                   batch_size, shuffle=True,
                                   num_workers=get_dataloader_workers())
```

Let's look at the time it takes to read the training data.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
'%.2f sec' % timer.stop()
```

```
'1.65 sec'
```

3.5.3 Putting All Things Together

Now we define the `load_data_fashion_mnist` function that obtains and reads the Fashion-MNIST dataset. It returns the data iterators for both the training set and validation set. In addition, it accepts an optional argument to resize images to another shape.

```
# Saved in the d2l package for later use
def load_data_fashion_mnist(batch_size, resize=None):
    """Download the Fashion-MNIST dataset and then load into memory."""
    dataset = gluon.data.vision
    trans = [dataset.transforms.Resize(resize)] if resize else []
    trans.append(dataset.transforms.ToTensor())
    trans = dataset.transforms.Compose(trans)
    mnist_train = dataset.FashionMNIST(train=True).transform_first(trans)
    mnist_test = dataset.FashionMNIST(train=False).transform_first(trans)
    return (gluon.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers()),
            gluon.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                  num_workers=get_dataloader_workers()))
```

Below, we verify that image resizing works.

```
train_iter, test_iter = load_data_fashion_mnist(32, (64, 64))
for X, y in train_iter:
    print(X.shape)
    break
```

```
(32, 1, 64, 64)
```

We are now ready to work with the FashionMNIST dataset in the sections that follow.

Summary

- Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories.
- We will use this dataset in subsequent sections and chapters to evaluate various classification algorithms.
- We store the shape of each image with height h width w pixels as $h \times w$ or `(h, w)`.
- Data iterators are a key component for efficient performance. Rely on well-implemented iterators that exploit multi-threading to avoid slowing down your training loop.

Exercises

1. Does reducing the `batch_size` (for instance, to 1) affect read performance?
2. For non-Windows users, try modifying `num_workers` to see how it affects read performance. Plot the performance against the number of works employed.
3. Use the MXNet documentation to see which other datasets are available in `mxnet.gluon.data.vision`.
4. Use the MXNet documentation to see which other transformations are available in `mxnet.gluon.data.vision.transforms`.

3.6 Implementation of Softmax Regression from Scratch

Just as we implemented linear regression from scratch, we believe that multiclass logistic (softmax) regression is similarly fundamental and you ought to know the gory details of how to implement it yourself. As with linear regression, after doing things by hand we will breeze through an implementation in Gluon for comparison. To begin, let's import the familiar packages.

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

We will work with the Fashion-MNIST dataset, just introduced in Section 3.5, setting up an iterator with batch size 256.

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

3.6.1 Initializing Model Parameters

As in our linear regression example, each example here will be represented by a fixed-length vector. Each example in the raw data is a 28×28 image. In this section, we will flatten each image, treating them as 784 1D vectors. In the future, we will talk about more sophisticated strategies for exploiting the spatial structure in images, but for now we treat each pixel location as just another feature.

Recall that in softmax regression, we have as many outputs as there are categories. Because our dataset has 10 categories, our network will have an output dimension of 10. Consequently, our weights will constitute a 784×10 matrix and the biases will constitute a 1×10 vector. As with linear regression, we will initialize our weights W with Gaussian noise and our biases to take the initial value 0.

```
num_inputs = 784
num_outputs = 10

W = np.random.normal(0, 0.01, (num_inputs, num_outputs))
b = np.zeros(num_outputs)
```

Recall that we need to *attach gradients* to the model parameters. More literally, we are allocating memory for future gradients to be stored and notifiying MXNet that we will want to calculate gradients with respect to these parameters in the future.

```
W.attach_grad()
b.attach_grad()
```

3.6.2 The Softmax

Before implementing the softmax regression model, let's briefly review how operators such as `sum` work along specific dimensions in an `ndarray`. Given a matrix `X` we can sum over all elements (default) or only over elements in the same axis, *i.e.*, the column (`axis=0`) or the same row (`axis=1`). Note that if `X` is an array with shape `(2, 3)` and we sum over the columns (`X.sum(axis=0)`, the result will be a (1D) vector with shape `(3,)`. If we want to keep the number of axes in the original array (resulting in a 2D array with shape `(1, 3)`), rather than collapsing out the dimension that we summed over we can specify `keepdims=True` when invoking `sum`.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
print(X.sum(axis=0, keepdims=True), '\n', X.sum(axis=1, keepdims=True))
```

```
[[5. 7. 9.]]
 [[ 6.]
 [15.]]
```

We are now ready to implement the softmax function. Recall that softmax consists of two steps: First, we exponentiate each term (using exp). Then, we sum over each row (we have one row per example in the batch) to get the normalization constants for each example. Finally, we divide each row by its normalization constant, ensuring that the result sums to 1. Before looking at the code, let's recall what this looks expressed as an equation:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}. \tag{3.6.1}$$

The denominator, or normalization constant, is also sometimes called the partition function (and its logarithm is called the log-partition function). The origins of that name are in statistical physics[59] where a related equation models the distribution over an ensemble of particles).

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

As you can see, for any random input, we turn each element into a non-negative number. Moreover, each row sums up to 1, as is required for a probability. Note that while this looks correct mathematically, we were a bit sloppy in our implementation because failed to take precautions against numerical overflow or underflow due to large (or very small) elements of the matrix, as we did in Section 17.8.

```
X = np.random.normal(size=(2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(axis=1)
```

```
(array([[0.22376052, 0.06659239, 0.06583703, 0.29964197, 0.3441681 ],
        [0.63209665, 0.03179282, 0.194987  , 0.09209415, 0.04902935]]),
 array([1.        , 0.99999994]))
```

3.6.3 The Model

Now that we have defined the softmax operation, we can implement the softmax regression model. The below code defines the forward pass through the network. Note that we flatten each original image in the batch into a vector with length num_inputs with the reshape function before passing the data through our model.

```
def net(X):
    return softmax(np.dot(X.reshape(-1, num_inputs), W) + b)
```

[59] https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics)

3.6.4 The Loss Function

Next, we need to implement the cross-entropy loss function, introduced in Section 3.4. This may be the most common loss function in all of deep learning because, at the moment, classification problems far outnumber regression problems.

Recall that cross-entropy takes the negative log likelihood of the predicted probability assigned to the true label $-\log P(y \mid x)$. Rather than iterating over the predictions with a Python `for` loop (which tends to be inefficient), we can use the `pick` function which allows us to easily select the appropriate terms from the matrix of softmax entries. Below, we illustrate the `pick` function on a toy example, with 3 categories and 2 examples.

```
y_hat = np.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], [0, 2]]
```

```
array([0.1, 0.5])
```

Now we can implement the cross-entropy loss function efficiently with just one line of code.

```
def cross_entropy(y_hat, y):
    return - np.log(y_hat[range(len(y_hat)), y])
```

3.6.5 Classification Accuracy

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

When predictions are consistent with the actual category `y`, they are correct. The classification accuracy is the fraction of all predictions that are correct. Although it can be difficult optimize accuracy directly (it is not differentiable), it is often the performance metric that we care most about, and we will nearly always report it when training classifiers.

To compute accuracy we do the following: First, we execute `y_hat.argmax(axis=1)` to gather the predicted classes (given by the indices for the largest entires each row). The result has the same shape as the variable `y`. Now we just need to check how frequently the two match. Since the equality operator == is datatype-sensitive (e.g., an `int` and a `float32` are never equal), we also need to convert both to the same type (we pick `float32`). The result is an `ndarray` containing entries of 0 (false) and 1 (true). Taking the mean yields the desired result.

```
# Saved in the d2l package for later use
def accuracy(y_hat, y):
    if y_hat.shape[1] > 1:
        return float((y_hat.argmax(axis=1) == y.astype('float32')).sum())
    else:
        return float((y_hat.astype('int32') == y.astype('int32')).sum())
```

We will continue to use the variables `y_hat` and `y` defined in the `pick` function, as the predicted probability distribution and label, respectively. We can see that the first example's prediction category is 2 (the largest element of the row is 0.6 with an index of 2), which is inconsistent with the

actual label, 0. The second example's prediction category is 2 (the largest element of the row is 0.5 with an index of 2), which is consistent with the actual label, 2. Therefore, the classification accuracy rate for these two examples is 0.5.

```
y = np.array([0, 2])
accuracy(y_hat, y) / len(y)
```

```
0.5
```

Similarly, we can evaluate the accuracy for model `net` on the dataset (accessed via `data_iter`).

```
# Saved in the d2l package for later use
def evaluate_accuracy(net, data_iter):
    metric = Accumulator(2)  # num_corrected_examples, num_examples
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), y.size)
    return metric[0] / metric[1]
```

Here `Accumulator` is a utility class to accumulated sum over multiple numbers.

```
# Saved in the d2l package for later use
class Accumulator(object):
    """Sum a list of numbers over time"""

    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a+b for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0] * len(self.data)

    def __getitem__(self, i):
        return self.data[i]
```

Because we initialized the `net` model with random weights, the accuracy of this model should be close to random guessing, i.e., 0.1 for 10 classes.

```
evaluate_accuracy(net, test_iter)
```

```
0.0811
```

3.6.6 Model Training

The training loop for softmax regression should look strikingly familiar if you read through our implementation of linear regression in Section 3.2. Here we refactor the implementation to make it reusable. First, we define a function to train for one data epoch. Note that `updater` is general function to update the model parameters, which accepts the batch size as an argument. It can be either a wrapper of `d2l.sgd` or a Gluon trainer.

```
# Saved in the d2l package for later use
def train_epoch_ch3(net, train_iter, loss, updater):
    metric = Accumulator(3)  # train_loss_sum, train_acc_sum, num_examples
    if isinstance(updater, gluon.Trainer):
        updater = updater.step
    for X, y in train_iter:
        # Compute gradients and update parameters
        with autograd.record():
            y_hat = net(X)
            l = loss(y_hat, y)
        l.backward()
        updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.size)
    # Return training loss and training accuracy
    return metric[0]/metric[2], metric[1]/metric[2]
```

Before showing the implementation of the training function, we define a utility class that draw data in animation. Again, it aims to simplify the codes in later chapters.

```
# Saved in the d2l package for later use
class Animator(object):
    def __init__(self, xlabel=None, ylabel=None, legend=[], xlim=None,
                 ylim=None, xscale='linear', yscale='linear', fmts=None,
                 nrows=1, ncols=1, figsize=(3.5, 2.5)):
        """Incrementally plot multiple lines."""
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a lambda to capture arguments
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        """Add multiple data points into the figure."""
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        if not self.fmts:
            self.fmts = ['-'] * n
```

(continues on next page)

(continued from previous page)

```
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
```

The training function then runs multiple epochs and visualize the training progress.

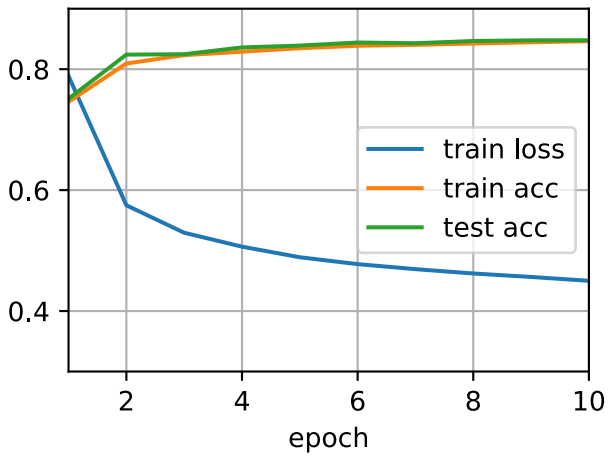
```
# Saved in the d2l package for later use
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs],
                        ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch+1, train_metrics+(test_acc,))
```

Again, we use the minibatch stochastic gradient descent to optimize the loss function of the model. Note that the number of epochs (`num_epochs`), and learning rate (`lr`) are both adjustable hyperparameters. By changing their values, we may be able to increase the classification accuracy of the model. In practice we will want to split our data three ways into training, validation, and test data, using the validation data to choose the best values of our hyperparameters.

```
num_epochs, lr = 10, 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)

train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```

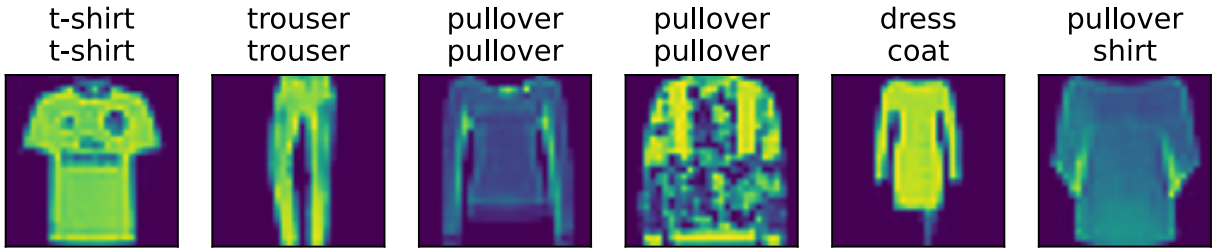


3.6.7 Prediction

Now that training is complete, our model is ready to classify some images. Given a series of images, we will compare their actual labels (first line of text output) and the model predictions (second line of text output).

```
# Saved in the d2l package for later use
def predict_ch3(net, test_iter, n=6):
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true+'\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(X[0:n].reshape(n, 28, 28), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```



Summary

With softmax regression, we can train models for multi-category classification. The training loop is very similar to that in linear regression: retrieve and read data, define models and loss functions, then train models using optimization algorithms. As you will soon find out, most common deep learning models have similar training procedures.

Exercises

1. In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. What problems might this cause (hint: try to calculate the size of $\exp(50)$)?
2. The function `cross_entropy` in this section is implemented according to the definition of the cross-entropy loss function. What could be the problem with this implementation (hint: consider the domain of the logarithm)?
3. What solutions you can think of to fix the two problems above?
4. Is it always a good idea to return the most likely label. E.g. would you do this for medical diagnosis?
5. Assume that we want to use softmax regression to predict the next word based on some features. What are some problems that might arise from a large vocabulary?

3.7 Concise Implementation of Softmax Regression

Just as Gluon made it much easier to implement linear regression in Section 3.3, we will find it similarly (or possibly more) convenient for implementing classification models. Again, we begin with our import ritual.

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

Let's stick with the Fashion-MNIST dataset and keep the batch size at 256 as in the last section.

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

3.7.1 Initializing Model Parameters

As mentioned in Section 3.4, the output layer of softmax regression is a fully-connected (`Dense`) layer. Therefore, to implement our model, we just need to add one `Dense` layer with 10 outputs to our `Sequential`. Again, here, the `Sequential` is not really necessary, but we might as well form the habit since it will be ubiquitous when implementing deep models. Again, we initialize the weights at random with zero mean and standard deviation 0.01.

```
net = nn.Sequential()
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

3.7.2 The Softmax

In the previous example, we calculated our model's output and then ran this output through the cross-entropy loss. Mathematically, that is a perfectly reasonable thing to do. However, from a computational perspective, exponentiation can be a source of numerical stability issues (as discussed in Section 17.8). Recall that the softmax function calculates $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}$, where $\hat{y}_j$ is the j^{th} element of `yhat` and z_j is the j^{th} element of the input `y_linear` variable, as computed by the softmax.

If some of the z_i are very large (i.e., very positive), then e^{z_i} might be larger than the largest number we can have for certain types of `float` (i.e., overflow). This would make the denominator (and/or numerator) `inf` and we wind up encountering either 0, `inf`, or `nan` for $\hat{y}_j$. In these situations we do not get a well-defined return value for `cross_entropy`. One trick to get around this is to first

subtract $\max(z_i)$ from all z_i before proceeding with the `softmax` calculation. You can verify that this shifting of each z_i by constant factor does not change the return value of `softmax`.

After the subtraction and normalization step, it might be that possible that some z_j have large negative values and thus that the corresponding e^{z_j} will take values close to zero. These might be rounded to zero due to finite precision (i.e underflow), making $\hat{y}_j$ zero and giving us `-inf` for $\log(\hat{y}_j)$. A few steps down the road in backpropagation, we might find ourselves faced with a screenful of the dreaded not-a-number (`nan`) results.

Fortunately, we are saved by the fact that even though we are computing exponential functions, we ultimately intend to take their log (when calculating the cross-entropy loss). By combining these two operators (`softmax` and `cross_entropy`) together, we can escape the numerical stability issues that might otherwise plague us during backpropagation. As shown in the equation below, we avoided calculating e^{z_j} and can instead z_j directly due to the canceling in $\log(\exp(\cdot))$.

$$\begin{aligned} \log\left(\hat{y}_j\right) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}\right) \\ &= \log\left(e^{z_j}\right) - \log\left(\sum_{i=1}^{n} e^{z_i}\right) \\ &= z_j - \log\left(\sum_{i=1}^{n} e^{z_i}\right). \end{aligned} \tag{3.7.1}$$

We will want to keep the conventional softmax function handy in case we ever want to evaluate the probabilities output by our model. But instead of passing softmax probabilities into our new loss function, we will just pass the logits and compute the softmax and its log all at once inside the softmax_cross_entropy loss function, which does smart things like the log-sum-exp trick (see on Wikipedia[61]).

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

3.7.3 Optimization Algorithm

Here, we use minibatch stochastic gradient descent with a learning rate of 0.1 as the optimization algorithm. Note that this is the same as we applied in the linear regression example and it illustrates the general applicability of the optimizers.

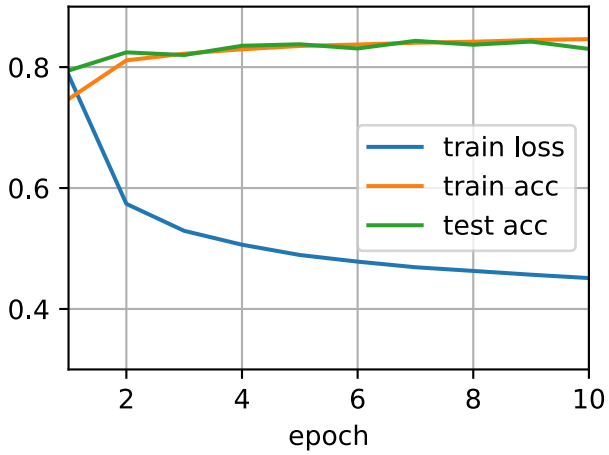
```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

3.7.4 Training

Next we call the training function defined in the last section to train a model.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

[61] https://en.wikipedia.org/wiki/LogSumExp



As before, this algorithm converges to a solution that achieves an accuracy of 83.7%, albeit this time with fewer lines of code than before. Note that in many cases, Gluon takes additional precautions beyond these most well-known tricks to ensure numerical stability, saving us from even more pitfalls that we would encounter if we tried to code all of our models from scratch in practice.

Exercises

1. Try adjusting the hyper-parameters, such as batch size, epoch, and learning rate, to see what the results are.
2. Why might the test accuracy decrease again after a while? How could we fix this?