Chapter 19. Deep Learning

> *A little learning is a dangerous thing; Drink deep, or taste not the Pierian spring.*
>
> —Alexander Pope

Deep learning originally referred to the application of "deep" neural networks (that is, networks with more than one hidden layer), although in practice the term now encompasses a wide variety of neural architectures (including the "simple" neural networks we developed in Chapter 18).

In this chapter we'll build on our previous work and look at a wider variety of neural networks. To do so, we'll introduce a number of abstractions that allow us to think about neural networks in a more general way.

The Tensor

Previously, we made a distinction between vectors (one-dimensional arrays) and matrices (two-dimensional arrays). When we start working with more complicated neural networks, we'll need to use higher-dimensional arrays as well.

In many neural network libraries, *n*-dimensional arrays are referred to as *tensors*, which is what we'll call them too. (There are pedantic mathematical reasons not to refer to *n*-dimensional arrays as tensors; if you are such a pedant, your objection is noted.)

If I were writing an entire book about deep learning, I'd implement a full-featured `Tensor` class that overloaded Python's arithmetic operators and could handle a variety of other operations. Such an implementation would take an entire chapter on its own. Here we'll cheat and say that a `Tensor` is just a `list`. This is true in one direction—all of our vectors and matrices and higher-dimensional analogues *are* lists. It is certainly not true in the

other direction—most Python `lists` are not *n*-dimensional arrays in our sense.

> **NOTE**
>
> Ideally you'd like to do something like:
>
> ```
> # A Tensor is either a float, or a List of Tensors
> Tensor = Union[float, List[Tensor]]
> ```
>
> However, Python won't let you define recursive types like that. And even if it did that definition is still not right, as it allows for bad "tensors" like:
>
> ```
> [[1.0, 2.0],
>  [3.0]]
> ```
>
> whose rows have different sizes, which makes it not an *n*-dimensional array.

So, like I said, we'll just cheat:

```
Tensor = list
```

And we'll write a helper function to find a tensor's *shape*:

```
from typing import List

def shape(tensor: Tensor) -> List[int]:
    sizes: List[int] = []
    while isinstance(tensor, list):
        sizes.append(len(tensor))
        tensor = tensor[0]
    return sizes

assert shape([1, 2, 3]) == [3]
assert shape([[1, 2], [3, 4], [5, 6]]) == [3, 2]
```

Because tensors can have any number of dimensions, we'll typically need to work with them recursively. We'll do one thing in the one-dimensional

case and recurse in the higher-dimensional case:

```
def is_1d(tensor: Tensor) -> bool:
    """
    If tensor[0] is a list, it's a higher-order tensor.
    Otherwise, tensor is 1-dimensional (that is, a vector).
    """
    return not isinstance(tensor[0], list)

assert is_1d([1, 2, 3])
assert not is_1d([[1, 2], [3, 4]])
```

which we can use to write a recursive `tensor_sum` function:

```
def tensor_sum(tensor: Tensor) -> float:
    """Sums up all the values in the tensor"""
    if is_1d(tensor):
        return sum(tensor)  # just a list of floats, use Python sum
    else:
        return sum(tensor_sum(tensor_i)      # Call tensor_sum on each row
                   for tensor_i in tensor)   # and sum up those results.

assert tensor_sum([1, 2, 3]) == 6
assert tensor_sum([[1, 2], [3, 4]]) == 10
```

If you're not used to thinking recursively, you should ponder this until it makes sense, because we'll use the same logic throughout this chapter. However, we'll create a couple of helper functions so that we don't have to rewrite this logic everywhere. The first applies a function elementwise to a single tensor:

```
from typing import Callable

def tensor_apply(f: Callable[[float], float], tensor: Tensor) -> Tensor:
    """Applies f elementwise"""
    if is_1d(tensor):
        return [f(x) for x in tensor]
    else:
        return [tensor_apply(f, tensor_i) for tensor_i in tensor]

assert tensor_apply(lambda x: x + 1, [1, 2, 3]) == [2, 3, 4]
assert tensor_apply(lambda x: 2 * x, [[1, 2], [3, 4]]) == [[2, 4], [6, 8]]
```

We can use this to write a function that creates a zero tensor with the same shape as a given tensor:

```
def zeros_like(tensor: Tensor) -> Tensor:
    return tensor_apply(lambda _: 0.0, tensor)

assert zeros_like([1, 2, 3]) == [0, 0, 0]
assert zeros_like([[1, 2], [3, 4]]) == [[0, 0], [0, 0]]
```

We'll also need to apply a function to corresponding elements from two tensors (which had better be the exact same shape, although we won't check that):

```
def tensor_combine(f: Callable[[float, float], float],
                   t1: Tensor,
                   t2: Tensor) -> Tensor:
    """Applies f to corresponding elements of t1 and t2"""
    if is_1d(t1):
        return [f(x, y) for x, y in zip(t1, t2)]
    else:
        return [tensor_combine(f, t1_i, t2_i)
                for t1_i, t2_i in zip(t1, t2)]

import operator
assert tensor_combine(operator.add, [1, 2, 3], [4, 5, 6]) == [5, 7, 9]
assert tensor_combine(operator.mul, [1, 2, 3], [4, 5, 6]) == [4, 10, 18]
```

The Layer Abstraction

In the previous chapter we built a simple neural net that allowed us to stack two layers of neurons, each of which computed `sigmoid(dot(weights, inputs))`.

Although that's perhaps an idealized representation of what an actual neuron does, in practice we'd like to allow a wider variety of things. Perhaps we'd like the neurons to remember something about their previous inputs. Perhaps we'd like to use a different activation function than `sigmoid`. And frequently we'd like to use more than two layers. (Our

`feed_forward` function actually handled any number of layers, but our gradient computations did not.)

In this chapter we'll build machinery for implementing such a variety of neural networks. Our fundamental abstraction will be the `Layer`, something that knows how to apply some function to its inputs and that knows how to backpropagate gradients.

One way of thinking about the neural networks we built in Chapter 18 is as a "linear" layer, followed by a "sigmoid" layer, then another linear layer and another sigmoid layer. We didn't distinguish them in these terms, but doing so will allow us to experiment with much more general structures:

```
from typing import Iterable, Tuple

class Layer:
    """
    Our neural networks will be composed of Layers, each of which
    knows how to do some computation on its inputs in the "forward"
    direction and propagate gradients in the "backward" direction.
    """
    def forward(self, input):
        """
        Note the lack of types. We're not going to be prescriptive
        about what kinds of inputs layers can take and what kinds
        of outputs they can return.
        """
        raise NotImplementedError

    def backward(self, gradient):
        """
        Similarly, we're not going to be prescriptive about what the
        gradient looks like. It's up to you the user to make sure
        that you're doing things sensibly.
        """
        raise NotImplementedError

    def params(self) -> Iterable[Tensor]:
        """
        Returns the parameters of this layer. The default implementation
        returns nothing, so that if you have a layer with no parameters
        you don't have to implement this.
        """
        return ()
```

```
    def grads(self) -> Iterable[Tensor]:
        """
        Returns the gradients, in the same order as params().
        """
        return ()
```

The `forward` and `backward` methods will have to be implemented in our concrete subclasses. Once we build a neural net, we'll want to train it using gradient descent, which means we'll want to update each parameter in the network using its gradient. Accordingly, we insist that each layer be able to tell us its parameters and gradients.

Some layers (for example, a layer that applies `sigmoid` to each of its inputs) have no parameters to update, so we provide a default implementation that handles that case.

Let's look at that layer:

```
from scratch.neural_networks import sigmoid

class Sigmoid(Layer):
    def forward(self, input: Tensor) -> Tensor:
        """
        Apply sigmoid to each element of the input tensor,
        and save the results to use in backpropagation.
        """
        self.sigmoids = tensor_apply(sigmoid, input)
        return self.sigmoids

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda sig, grad: sig * (1 - sig) * grad,
                              self.sigmoids,
                              gradient)
```

There are a couple of things to notice here. One is that during the forward pass we saved the computed sigmoids so that we could use them later in the backward pass. Our layers will typically need to do this sort of thing.

Second, you may be wondering where the `sig * (1 - sig) * grad` comes from. This is just the chain rule from calculus and corresponds to the

`output * (1 - output) * (output - target)` term in our previous neural networks.

Finally, you can see how we were able to make use of the `tensor_apply` and the `tensor_combine` functions. Most of our layers will use these functions similarly.

The Linear Layer

The other piece we'll need to duplicate the neural networks from Chapter 18 is a "linear" layer that represents the `dot(weights, inputs)` part of the neurons.

This layer will have parameters, which we'd like to initialize with random values.

It turns out that the initial parameter values can make a huge difference in how quickly (and sometimes *whether*) the network trains. If weights are too big, they may produce large outputs in a range where the activation function has near-zero gradients. And parts of the network that have zero gradients necessarily can't learn anything via gradient descent.

Accordingly, we'll implement three different schemes for randomly generating our weight tensors. The first is to choose each value from the random uniform distribution on [0, 1]—that is, as a `random.random()`. The second (and default) is to choose each value randomly from a standard normal distribution. And the third is to use *Xavier initialization*, where each weight is initialized with a random draw from a normal distribution with mean 0 and variance 2 / (`num_inputs` + `num_outputs`). It turns out this often works nicely for neural network weights. We'll implement these with a `random_uniform` function and a `random_normal` function:

```
import random

from scratch.probability import inverse_normal_cdf

def random_uniform(*dims: int) -> Tensor:
```

```
    if len(dims) == 1:
        return [random.random() for _ in range(dims[0])]
    else:
        return [random_uniform(*dims[1:]) for _ in range(dims[0])]

def random_normal(*dims: int,
                  mean: float = 0.0,
                  variance: float = 1.0) -> Tensor:
    if len(dims) == 1:
        return [mean + variance * inverse_normal_cdf(random.random())
                for _ in range(dims[0])]
    else:
        return [random_normal(*dims[1:], mean=mean, variance=variance)
                for _ in range(dims[0])]

assert shape(random_uniform(2, 3, 4)) == [2, 3, 4]
assert shape(random_normal(5, 6, mean=10)) == [5, 6]
```

And then wrap them all in a `random_tensor` function:

```
def random_tensor(*dims: int, init: str = 'normal') -> Tensor:
    if init == 'normal':
        return random_normal(*dims)
    elif init == 'uniform':
        return random_uniform(*dims)
    elif init == 'xavier':
        variance = len(dims) / sum(dims)
        return random_normal(*dims, variance=variance)
    else:
        raise ValueError(f"unknown init: {init}")
```

Now we can define our linear layer. We need to initialize it with the dimension of the inputs (which tells us how many weights each neuron needs), the dimension of the outputs (which tells us how many neurons we should have), and the initialization scheme we want:

```
from scratch.linear_algebra import dot

class Linear(Layer):
    def __init__(self,
                 input_dim: int,
                 output_dim: int,
                 init: str = 'xavier') -> None:
        """
```

```
        A layer of output_dim neurons, each with input_dim weights
        (and a bias).
        """
        self.input_dim = input_dim
        self.output_dim = output_dim

        # self.w[o] is the weights for the oth neuron
        self.w = random_tensor(output_dim, input_dim, init=init)

        # self.b[o] is the bias term for the oth neuron
        self.b = random_tensor(output_dim, init=init)
```

> **NOTE**
>
> In case you're wondering how important the initialization schemes are, some of the networks in this chapter I couldn't get to train at all with different initializations than the ones I used.

The `forward` method is easy to implement. We'll get one output per neuron, which we stick in a vector. And each neuron's output is just the `dot` of its weights with the input, plus its bias:

```
    def forward(self, input: Tensor) -> Tensor:
        # Save the input to use in the backward pass.
        self.input = input

        # Return the vector of neuron outputs.
        return [dot(input, self.w[o]) + self.b[o]
                for o in range(self.output_dim)]
```

The `backward` method is more involved, but if you know calculus it's not difficult:

```
    def backward(self, gradient: Tensor) -> Tensor:
        # Each b[o] gets added to output[o], which means
        # the gradient of b is the same as the output gradient.
        self.b_grad = gradient

        # Each w[o][i] multiplies input[i] and gets added to output[o].
        # So its gradient is input[i] * gradient[o].
        self.w_grad = [[self.input[i] * gradient[o]
```

```
                        for i in range(self.input_dim)]
                    for o in range(self.output_dim)]

        # Each input[i] multiplies every w[o][i] and gets added to every
        # output[o]. So its gradient is the sum of w[o][i] * gradient[o]
        # across all the outputs.
        return [sum(self.w[o][i] * gradient[o] for o in
range(self.output_dim))
                for i in range(self.input_dim)]
```

> **NOTE**
>
> In a "real" tensor library, these (and many other) operations would be represented as matrix or tensor multiplications, which those libraries are designed to do very quickly. Our library is *very* slow.

Finally, here we do need to implement `params` and `grads`. We have two parameters and two corresponding gradients:

```
    def params(self) -> Iterable[Tensor]:
        return [self.w, self.b]

    def grads(self) -> Iterable[Tensor]:
        return [self.w_grad, self.b_grad]
```

Neural Networks as a Sequence of Layers

We'd like to think of neural networks as sequences of layers, so let's come up with a way to combine multiple layers into one. The resulting neural network is itself a layer, and it implements the `Layer` methods in the obvious ways:

```
from typing import List

class Sequential(Layer):
    """
    A layer consisting of a sequence of other layers.
    It's up to you to make sure that the output of each layer
    makes sense as the input to the next layer.
```

```
    """
    def __init__(self, layers: List[Layer]) -> None:
        self.layers = layers

    def forward(self, input):
        """Just forward the input through the layers in order."""
        for layer in self.layers:
            input = layer.forward(input)
        return input

    def backward(self, gradient):
        """Just backpropagate the gradient through the layers in reverse."""
        for layer in reversed(self.layers):
            gradient = layer.backward(gradient)
        return gradient

    def params(self) -> Iterable[Tensor]:
        """Just return the params from each layer."""
        return (param for layer in self.layers for param in layer.params())

    def grads(self) -> Iterable[Tensor]:
        """Just return the grads from each layer."""
        return (grad for layer in self.layers for grad in layer.grads())
```

So we could represent the neural network we used for XOR as:

```
xor_net = Sequential([
    Linear(input_dim=2, output_dim=2),
    Sigmoid(),
    Linear(input_dim=2, output_dim=1),
    Sigmoid()
])
```

But we still need a little more machinery to train it.

Loss and Optimization

Previously we wrote out individual loss functions and gradient functions for our models. Here we'll want to experiment with different loss functions, so (as usual) we'll introduce a new `Loss` abstraction that encapsulates both the loss computation and the gradient computation:

```
class Loss:
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        """How good are our predictions? (Larger numbers are worse.)"""
        raise NotImplementedError

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        """How does the loss change as the predictions change?"""
        raise NotImplementedError
```

We've already worked many times with the loss that's the sum of the squared errors, so we should have an easy time implementing that. The only trick is that we'll need to use `tensor_combine`:

```
class SSE(Loss):
    """Loss function that computes the sum of the squared errors."""
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        # Compute the tensor of squared differences
        squared_errors = tensor_combine(
            lambda predicted, actual: (predicted - actual) ** 2,
            predicted,
            actual)

        # And just add them up
        return tensor_sum(squared_errors)

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        return tensor_combine(
            lambda predicted, actual: 2 * (predicted - actual),
            predicted,
            actual)
```

(We'll look at a different loss function in a bit.)

The last piece to figure out is gradient descent. Throughout the book we've done all of our gradient descent manually by having a training loop that involves something like:

```
theta = gradient_step(theta, grad, -learning_rate)
```

Here that won't quite work for us, for a couple reasons. The first is that our neural nets will have many parameters, and we'll need to update all of

them. The second is that we'd like to be able to use more clever variants of gradient descent, and we don't want to have to rewrite them each time.

Accordingly, we'll introduce a (you guessed it) `Optimizer` abstraction, of which gradient descent will be a specific instance:

```
class Optimizer:
    """
    An optimizer updates the weights of a layer (in place) using information
    known by either the layer or the optimizer (or by both).
    """
    def step(self, layer: Layer) -> None:
        raise NotImplementedError
```

After that it's easy to implement gradient descent, again using `tensor_combine`:

```
class GradientDescent(Optimizer):
    def __init__(self, learning_rate: float = 0.1) -> None:
        self.lr = learning_rate

    def step(self, layer: Layer) -> None:
        for param, grad in zip(layer.params(), layer.grads()):
            # Update param using a gradient step
            param[:] = tensor_combine(
                lambda param, grad: param - grad * self.lr,
                param,
                grad)
```

The only thing that's maybe surprising is the "slice assignment," which is a reflection of the fact that reassigning a list doesn't change its original value. That is, if you just did `param = tensor_combine(. . .)`, you would be redefining the local variable `param`, but you would not be affecting the original parameter tensor stored in the layer. If you assign to the slice `[:]`, however, it actually changes the values inside the list.

Here's a simple example to demonstrate:

```
tensor = [[1, 2], [3, 4]]

for row in tensor:
```

```
    row = [0, 0]
assert tensor == [[1, 2], [3, 4]], "assignment doesn't update a list"

for row in tensor:
    row[:] = [0, 0]
assert tensor == [[0, 0], [0, 0]], "but slice assignment does"
```

If you are somewhat inexperienced in Python, this behavior may be surprising, so meditate on it and try examples yourself until it makes sense.

To demonstrate the value of this abstraction, let's implement another optimizer that uses *momentum*. The idea is that we don't want to overreact to each new gradient, and so we maintain a running average of the gradients we've seen, updating it with each new gradient and taking a step in the direction of the average:

```
class Momentum(Optimizer):
    def __init__(self,
                 learning_rate: float,
                 momentum: float = 0.9) -> None:
        self.lr = learning_rate
        self.mo = momentum
        self.updates: List[Tensor] = []  # running average

    def step(self, layer: Layer) -> None:
        # If we have no previous updates, start with all zeros
        if not self.updates:
            self.updates = [zeros_like(grad) for grad in layer.grads()]

        for update, param, grad in zip(self.updates,
                                       layer.params(),
                                       layer.grads()):
            # Apply momentum
            update[:] = tensor_combine(
                lambda u, g: self.mo * u + (1 - self.mo) * g,
                update,
                grad)

            # Then take a gradient step
            param[:] = tensor_combine(
                lambda p, u: p - self.lr * u,
                param,
                update)
```

Because we used an `Optimizer` abstraction, we can easily switch between our different optimizers.

Example: XOR Revisited

Let's see how easy it is to use our new framework to train a network that can compute XOR. We start by re-creating the training data:

```
# training data
xs = [[0., 0], [0., 1], [1., 0], [1., 1]]
ys = [[0.], [1.], [1.], [0.]]
```

and then we define the network, although now we can leave off the last sigmoid layer:

```
random.seed(0)

net = Sequential([
    Linear(input_dim=2, output_dim=2),
    Sigmoid(),
    Linear(input_dim=2, output_dim=1)
])
```

We can now write a simple training loop, except that now we can use the abstractions of `Optimizer` and `Loss`. This allows us to easily try different ones:

```
import tqdm

optimizer = GradientDescent(learning_rate=0.1)
loss = SSE()

with tqdm.trange(3000) as t:
    for epoch in t:
        epoch_loss = 0.0

        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
```

```
            net.backward(gradient)

            optimizer.step(net)

        t.set_description(f"xor loss {epoch_loss:.3f}")
```

This should train quickly, and you should see the loss go down. And now we can inspect the weights:

```
for param in net.params():
    print(param)
```

For my network I find roughly:

```
hidden1 = -2.6 * x1 + -2.7 * x2 + 0.2  # NOR
hidden2 =  2.1 * x1 +  2.1 * x2 - 3.4  # AND
output =  -3.1 * h1 + -2.6 * h2 + 1.8  # NOR
```

So `hidden1` activates if neither input is 1. `hidden2` activates if both inputs are 1. And `output` activates if neither hidden output is 1—that is, if it's not the case that neither input is 1 and it's also not the case that both inputs are 1. Indeed, this is exactly the logic of XOR.

Notice that this network learned different features than the one we trained in Chapter 18, but it still manages to do the same thing.

Other Activation Functions

The `sigmoid` function has fallen out of favor for a couple of reasons. One reason is that `sigmoid(0)` equals 1/2, which means that a neuron whose inputs sum to 0 has a positive output. Another is that its gradient is very close to 0 for very large and very small inputs, which means that its gradients can get "saturated" and its weights can get stuck.

One popular replacement is `tanh` ("hyperbolic tangent"), which is a different sigmoid-shaped function that ranges from –1 to 1 and outputs 0 if

its input is 0. The derivative of `tanh(x)` is just `1 - tanh(x) ** 2`, which makes the layer easy to write:

```
import math

def tanh(x: float) -> float:
    # If x is very large or very small, tanh is (essentially) 1 or -1.
    # We check for this because, e.g., math.exp(1000) raises an error.
    if x < -100:  return -1
    elif x > 100: return 1

    em2x = math.exp(-2 * x)
    return (1 - em2x) / (1 + em2x)

class Tanh(Layer):
    def forward(self, input: Tensor) -> Tensor:
        # Save tanh output to use in backward pass.
        self.tanh = tensor_apply(tanh, input)
        return self.tanh

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(
            lambda tanh, grad: (1 - tanh ** 2) * grad,
            self.tanh,
            gradient)
```

In larger networks another popular replacement is `Relu`, which is 0 for negative inputs and the identity for positive inputs:

```
class Relu(Layer):
    def forward(self, input: Tensor) -> Tensor:
        self.input = input
        return tensor_apply(lambda x: max(x, 0), input)

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda x, grad: grad if x > 0 else 0,
                              self.input,
                              gradient)
```

There are many others. I encourage you to play around with them in your networks.

Example: FizzBuzz Revisited

We can now use our "deep learning" framework to reproduce our solution from "Example: Fizz Buzz". Let's set up the data:

```
from scratch.neural_networks import binary_encode, fizz_buzz_encode, argmax

xs = [binary_encode(n) for n in range(101, 1024)]
ys = [fizz_buzz_encode(n) for n in range(101, 1024)]
```

and create the network:

```
NUM_HIDDEN = 25

random.seed(0)

net = Sequential([
    Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
    Tanh(),
    Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform'),
    Sigmoid()
])
```

As we're training, let's also track our accuracy on the training set:

```
def fizzbuzz_accuracy(low: int, hi: int, net: Layer) -> float:
    num_correct = 0
    for n in range(low, hi):
        x = binary_encode(n)
        predicted = argmax(net.forward(x))
        actual = argmax(fizz_buzz_encode(n))
        if predicted == actual:
            num_correct += 1

    return num_correct / (hi - low)


optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SSE()

with tqdm.trange(1000) as t:
    for epoch in t:
        epoch_loss = 0.0
```

```
            for x, y in zip(xs, ys):
                predicted = net.forward(x)
                epoch_loss += loss.loss(predicted, y)
                gradient = loss.gradient(predicted, y)
                net.backward(gradient)

                optimizer.step(net)

            accuracy = fizzbuzz_accuracy(101, 1024, net)
            t.set_description(f"fb loss: {epoch_loss:.2f} acc: {accuracy:.2f}")

    # Now check results on the test set
    print("test results", fizzbuzz_accuracy(1, 101, net))
```

After 1,000 training iterations, the model gets 90% accuracy on the test set; if you keep training it longer, it should do even better. (I don't think it's possible to train to 100% accuracy with only 25 hidden units, but it's definitely possible if you go up to 50 hidden units.)

Softmaxes and Cross-Entropy

The neural net we used in the previous section ended in a `Sigmoid` layer, which means that its output was a vector of numbers between 0 and 1. In particular, it could output a vector that was entirely 0s, or it could output a vector that was entirely 1s. Yet when we're doing classification problems, we'd like to output a 1 for the correct class and a 0 for all the incorrect classes. Generally our predictions will not be so perfect, but we'd at least like to predict an actual probability distribution over the classes.

For example, if we have two classes, and our model outputs `[0, 0]`, it's hard to make much sense of that. It doesn't think the output belongs in either class?

But if our model outputs `[0.4, 0.6]`, we can interpret it as a prediction that there's a probability of 0.4 that our input belongs to the first class and 0.6 that our input belongs to the second class.

In order to accomplish this, we typically forgo the final `Sigmoid` layer and instead use the `softmax` function, which converts a vector of real numbers

to a vector of probabilities. We compute `exp(x)` for each number in the vector, which results in a vector of positive numbers. After that, we just divide each of those positive numbers by the sum, which gives us a bunch of positive numbers that add up to 1—that is, a vector of probabilities.

If we ever end up trying to compute, say, `exp(1000)` we will get a Python error, so before taking the `exp` we subtract off the largest value. This turns out to result in the same probabilities; it's just safer to compute in Python:

```
def softmax(tensor: Tensor) -> Tensor:
    """Softmax along the last dimension"""
    if is_1d(tensor):
        # Subtract largest value for numerical stability.
        largest = max(tensor)
        exps = [math.exp(x - largest) for x in tensor]

        sum_of_exps = sum(exps)                 # This is the total "weight."
        return [exp_i / sum_of_exps             # Probability is the fraction
                for exp_i in exps]              # of the total weight.
    else:
        return [softmax(tensor_i) for tensor_i in tensor]
```

Once our network produces probabilities, we often use a different loss function called *cross-entropy* (or sometimes "negative log likelihood").

You may recall that in "Maximum Likelihood Estimation", we justified the use of least squares in linear regression by appealing to the fact that (under certain assumptions) the least squares coefficients maximized the likelihood of the observed data.

Here we can do something similar: if our network outputs are probabilities, the cross-entropy loss represents the negative log likelihood of the observed data, which means that minimizing that loss is the same as maximizing the log likelihood (and hence the likelihood) of the training data.

Typically we won't include the `softmax` function as part of the neural network itself. This is because it turns out that if `softmax` is part of your loss function but not part of the network itself, the gradients of the loss with respect to the network outputs are very easy to compute.

```python
class SoftmaxCrossEntropy(Loss):
    """
    This is the negative-log-likelihood of the observed values, given the
    neural net model. So if we choose weights to minimize it, our model will
    be maximizing the likelihood of the observed data.
    """
    def loss(self, predicted: Tensor, actual: Tensor) -> float:
        # Apply softmax to get probabilities
        probabilities = softmax(predicted)

        # This will be log p_i for the actual class i and 0 for the other
        # classes. We add a tiny amount to p to avoid taking log(0).
        likelihoods = tensor_combine(lambda p, act: math.log(p + 1e-30) * act,
                                     probabilities,
                                     actual)

        # And then we just sum up the negatives.
        return -tensor_sum(likelihoods)

    def gradient(self, predicted: Tensor, actual: Tensor) -> Tensor:
        probabilities = softmax(predicted)

        # Isn't this a pleasant equation?
        return tensor_combine(lambda p, actual: p - actual,
                              probabilities,
                              actual)
```

If I now train the same Fizz Buzz network using `SoftmaxCrossEntropy` loss, I find that it typically trains much faster (that is, in many fewer epochs). Presumably this is because it is much easier to find weights that `softmax` to a given distribution than it is to find weights that `sigmoid` to a given distribution.

That is, if I need to predict class 0 (a vector with a 1 in the first position and 0s in the remaining positions), in the `linear` + `sigmoid` case I need the first output to be a large positive number and the remaining outputs to be large negative numbers. In the `softmax` case, however, I just need the first output to be *larger than* the remaining outputs. Clearly there are a lot more ways for the second case to happen, which suggests that it should be easier to find weights that make it so:

```
random.seed(0)

net = Sequential([
    Linear(input_dim=10, output_dim=NUM_HIDDEN, init='uniform'),
    Tanh(),
    Linear(input_dim=NUM_HIDDEN, output_dim=4, init='uniform')
    # No final sigmoid layer now
])

optimizer = Momentum(learning_rate=0.1, momentum=0.9)
loss = SoftmaxCrossEntropy()

with tqdm.trange(100) as t:
    for epoch in t:
        epoch_loss = 0.0

        for x, y in zip(xs, ys):
            predicted = net.forward(x)
            epoch_loss += loss.loss(predicted, y)
            gradient = loss.gradient(predicted, y)
            net.backward(gradient)

            optimizer.step(net)

        accuracy = fizzbuzz_accuracy(101, 1024, net)
        t.set_description(f"fb loss: {epoch_loss:.3f} acc: {accuracy:.2f}")

# Again check results on the test set
print("test results", fizzbuzz_accuracy(1, 101, net))
```

Dropout

Like most machine learning models, neural networks are prone to overfitting to their training data. We've previously seen ways to ameliorate this; for example, in "Regularization" we penalized large weights and that helped prevent overfitting.

A common way of regularizing neural networks is using *dropout*. At training time, we randomly turn off each neuron (that is, replace its output with 0) with some fixed probability. This means that the network can't learn to depend on any individual neuron, which seems to help with overfitting.

At evaluation time, we don't want to dropout any neurons, so a `Dropout` layer will need to know whether it's training or not. In addition, at training time a `Dropout` layer only passes on some random fraction of its input. To make its output comparable during evaluation, we'll scale down the outputs (uniformly) using that same fraction:

```
class Dropout(Layer):
    def __init__(self, p: float) -> None:
        self.p = p
        self.train = True

    def forward(self, input: Tensor) -> Tensor:
        if self.train:
            # Create a mask of 0s and 1s shaped like the input
            # using the specified probability.
            self.mask = tensor_apply(
                lambda _: 0 if random.random() < self.p else 1,
                input)
            # Multiply by the mask to dropout inputs.
            return tensor_combine(operator.mul, input, self.mask)
        else:
            # During evaluation just scale down the outputs uniformly.
            return tensor_apply(lambda x: x * (1 - self.p), input)

    def backward(self, gradient: Tensor) -> Tensor:
        if self.train:
            # Only propagate the gradients where mask == 1.
            return tensor_combine(operator.mul, gradient, self.mask)
        else:
            raise RuntimeError("don't call backward when not in train mode")
```

We'll use this to help prevent our deep learning models from overfitting.

Example: MNIST

MNIST is a dataset of handwritten digits that everyone uses to learn deep learning.

It is available in a somewhat tricky binary format, so we'll install the `mnist` library to work with it. (Yes, this part is technically not "from scratch.")

```
python -m pip install mnist
```

And then we can load the data:

```
import mnist

# This will download the data; change this to where you want it.
# (Yes, it's a 0-argument function, that's what the library expects.)
# (Yes, I'm assigning a lambda to a variable, like I said never to do.)
mnist.temporary_dir = lambda: '/tmp'

# Each of these functions first downloads the data and returns a numpy array.
# We call .tolist() because our "tensors" are just lists.
train_images = mnist.train_images().tolist()
train_labels = mnist.train_labels().tolist()

assert shape(train_images) == [60000, 28, 28]
assert shape(train_labels) == [60000]
```

Let's plot the first 100 training images to see what they look like (Figure 19-1):

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(10, 10)

for i in range(10):
    for j in range(10):
        # Plot each image in black and white and hide the axes.
        ax[i][j].imshow(train_images[10 * i + j], cmap='Greys')
        ax[i][j].xaxis.set_visible(False)
        ax[i][j].yaxis.set_visible(False)

plt.show()
```

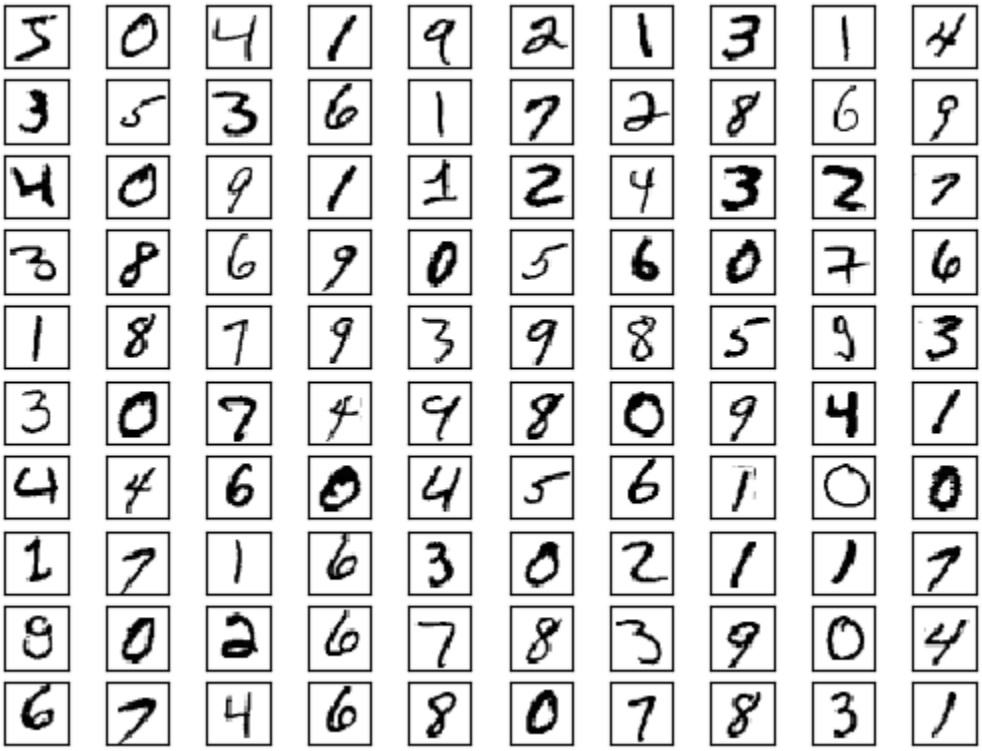

Figure 19-1. MNIST images

You can see that indeed they look like handwritten digits.

> **NOTE**
>
> My first attempt at showing the images resulted in yellow numbers on black backgrounds. I am neither clever nor subtle enough to know that I needed to add `cmap=Greys` to get black-and-white images; I Googled it and found the solution on Stack Overflow. As a data scientist you will become quite adept at this workflow.

We also need to load the test images:

```
test_images = mnist.test_images().tolist()
test_labels = mnist.test_labels().tolist()

assert shape(test_images) == [10000, 28, 28]
assert shape(test_labels) == [10000]
```

Each image is 28 × 28 pixels, but our linear layers can only deal with one-dimensional inputs, so we'll just flatten them (and also divide by 256 to get them between 0 and 1). In addition, our neural net will train better if our inputs are 0 on average, so we'll subtract out the average value:

```
# Compute the average pixel value
avg = tensor_sum(train_images) / 60000 / 28 / 28

# Recenter, rescale, and flatten
train_images = [[(pixel - avg) / 256 for row in image for pixel in row]
                for image in train_images]
test_images = [[(pixel - avg) / 256 for row in image for pixel in row]
               for image in test_images]

assert shape(train_images) == [60000, 784], "images should be flattened"
assert shape(test_images) == [10000, 784], "images should be flattened"

# After centering, average pixel should be very close to 0
assert -0.0001 < tensor_sum(train_images) < 0.0001
```

We also want to one-hot-encode the targets, since we have 10 outputs. First let's write a `one_hot_encode` function:

```
def one_hot_encode(i: int, num_labels: int = 10) -> List[float]:
    return [1.0 if j == i else 0.0 for j in range(num_labels)]

assert one_hot_encode(3) == [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
assert one_hot_encode(2, num_labels=5) == [0, 0, 1, 0, 0]
```

and then apply it to our data:

```
train_labels = [one_hot_encode(label) for label in train_labels]
test_labels = [one_hot_encode(label) for label in test_labels]

assert shape(train_labels) == [60000, 10]
assert shape(test_labels) == [10000, 10]
```

One of the strengths of our abstractions is that we can use the same training/evaluation loop with a variety of models. So let's write that first. We'll pass it our model, the data, a loss function, and (if we're training) an optimizer.

It will make a pass through our data, track performance, and (if we passed in an optimizer) update our parameters:

```
import tqdm

def loop(model: Layer,
         images: List[Tensor],
         labels: List[Tensor],
         loss: Loss,
         optimizer: Optimizer = None) -> None:
    correct = 0         # Track number of correct predictions.
    total_loss = 0.0    # Track total loss.

    with tqdm.trange(len(images)) as t:
        for i in t:
            predicted = model.forward(images[i])             # Predict.
            if argmax(predicted) == argmax(labels[i]):       # Check for
                correct += 1                                 # correctness.
            total_loss += loss.loss(predicted, labels[i])    # Compute loss.

            # If we're training, backpropagate gradient and update weights.
            if optimizer is not None:
                gradient = loss.gradient(predicted, labels[i])
                model.backward(gradient)
                optimizer.step(model)

            # And update our metrics in the progress bar.
            avg_loss = total_loss / (i + 1)
            acc = correct / (i + 1)
            t.set_description(f"mnist loss: {avg_loss:.3f} acc: {acc:.3f}")
```

As a baseline, we can use our deep learning library to train a (multiclass) logistic regression model, which is just a single linear layer followed by a softmax. This model (in essence) just looks for 10 linear functions such that if the input represents, say, a 5, then the 5th linear function produces the largest output.

One pass through our 60,000 training examples should be enough to learn the model:

```
random.seed(0)

# Logistic regression is just a linear layer followed by softmax
```

```
model = Linear(784, 10)
loss = SoftmaxCrossEntropy()

# This optimizer seems to work
optimizer = Momentum(learning_rate=0.01, momentum=0.99)

# Train on the training data
loop(model, train_images, train_labels, loss, optimizer)

# Test on the test data (no optimizer means just evaluate)
loop(model, test_images, test_labels, loss)
```

This gets about 89% accuracy. Let's see if we can do better with a deep neural network. We'll use two hidden layers, the first with 30 neurons, and the second with 10 neurons. And we'll use our `Tanh` activation:

```
random.seed(0)

# Name them so we can turn train on and off
dropout1 = Dropout(0.1)
dropout2 = Dropout(0.1)

model = Sequential([
    Linear(784, 30),  # Hidden layer 1: size 30
    dropout1,
    Tanh(),
    Linear(30, 10),   # Hidden layer 2: size 10
    dropout2,
    Tanh(),
    Linear(10, 10)    # Output layer: size 10
])
```

And we can just use the same training loop!

```
optimizer = Momentum(learning_rate=0.01, momentum=0.99)
loss = SoftmaxCrossEntropy()

# Enable dropout and train (takes > 20 minutes on my laptop!)
dropout1.train = dropout2.train = True
loop(model, train_images, train_labels, loss, optimizer)

# Disable dropout and evaluate
dropout1.train = dropout2.train = False
loop(model, test_images, test_labels, loss)
```

Our deep model gets better than 92% accuracy on the test set, which is a nice improvement from the simple logistic model.

The MNIST website describes a variety of models that outperform these. Many of them could be implemented using the machinery we've developed so far but would take an extremely long time to train in our lists-as-tensors framework. Some of the best models involve *convolutional* layers, which are important but unfortunately quite out of scope for an introductory book on data science.

Saving and Loading Models

These models take a long time to train, so it would be nice if we could save them so that we don't have to train them every time. Luckily, we can use the `json` module to easily serialize model weights to a file.

For saving, we can use `Layer.params` to collect the weights, stick them in a list, and use `json.dump` to save that list to a file:

```
import json

def save_weights(model: Layer, filename: str) -> None:
    weights = list(model.params())
    with open(filename, 'w') as f:
        json.dump(weights, f)
```

Loading the weights back is only a little more work. We just use `json.load` to get the list of weights back from the file and slice assignment to set the weights of our model.

(In particular, this means that we have to instantiate the model ourselves and *then* load the weights. An alternative approach would be to also save some representation of the model architecture and use that to instantiate the model. That's not a terrible idea, but it would require a lot more code and changes to all our `Layers`, so we'll stick with the simpler way.)

Before we load the weights, we'd like to check that they have the same shapes as the model params we're loading them into. (This is a safeguard against, for example, trying to load the weights for a saved deep network into a shallow network, or similar issues.)

```
def load_weights(model: Layer, filename: str) -> None:
    with open(filename) as f:
        weights = json.load(f)

    # Check for consistency
    assert all(shape(param) == shape(weight)
               for param, weight in zip(model.params(), weights))

    # Then load using slice assignment
    for param, weight in zip(model.params(), weights):
        param[:] = weight
```

> **NOTE**
>
> JSON stores your data as text, which makes it an extremely inefficient representation. In real applications you'd probably use the `pickle` serialization library, which serializes things to a more efficient binary format. Here I decided to keep it simple and human-readable.

You can download the weights for the various networks we train from the book's GitHub repository.

For Further Exploration

Deep learning is really hot right now, and in this chapter we barely scratched its surface. There are many good books and blog posts (and many, many bad blog posts) about almost any aspect of deep learning you'd like to know about.

- The canonical textbook *Deep Learning*, by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (MIT Press), is freely

available online. It is very good, but it involves quite a bit of mathematics.

- Francois Chollet's *Deep Learning with Python* (Manning) is a great introduction to the Keras library, after which our deep learning library is sort of patterned.
- I myself mostly use PyTorch for deep learning. Its website has lots of documentation and tutorials.