In [ ]:
```
#hide
! [ -e /content ] && pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

```
[[chapter_foundations]]
```

A Neural Net from the Foundations

This chapter begins a journey where we will dig deep into the internals of the models we used in the previous chapters. We will be covering many of the same things we've seen before, but this time around we'll be looking much more closely at the implementation details, and much less closely at the practical issues of how and why things are as they are.

We will build everything from scratch, only using basic indexing into a tensor. We'll write a neural net from the ground up, then implement backpropagation manually, so we know exactly what's happening in PyTorch when we call `loss.backward`. We'll also see how to extend PyTorch with custom *autograd* functions that allow us to specify our own forward and backward computations.

Building a Neural Net Layer from Scratch

Let's start by refreshing our understanding of how matrix multiplication is used in a basic neural network. Since we're building everything up from scratch, we'll use nothing but plain Python initially (except for indexing into PyTorch tensors), and then replace the plain Python with PyTorch functionality once we've seen how to create it.

Modeling a Neuron

A neuron receives a given number of inputs and has an internal weight for each of them. It sums those weighted inputs to produce an output and adds an inner bias. In math, this can be written as:

$$out = \sum_{i=1}^{n} x_i w_i + b$$

if we name our inputs $(x_1, \ldots, x_n)$, our weights $(w_1, \ldots, w_n)$, and our bias b. In code this translates into:

```
    output = sum([x*w for x,w in zip(inputs,weights)]) + bias
```

This output is then fed into a nonlinear function called an *activation function* before being sent to another neuron. In deep learning the most common of these is the *rectified Linear unit*, or *ReLU*, which, as we've seen, is a fancy way of saying:

```
    def relu(x): return x if x >= 0 else 0
```

A deep learning model is then built by stacking a lot of those neurons in successive layers. We create a first layer with a certain number of neurons (known as *hidden size*) and link all the inputs to each of those neurons. Such a layer is often called a *fully connected layer* or a *dense layer* (for densely connected), or a *linear layer*.

It requires to compute, for each `input` in our batch and each neuron with a give `weight`, the dot product:

```
    sum([x*w for x,w in zip(input,weight)])
```

If you have done a little bit of linear algebra, you may remember that having a lot of those dot products happens when you do a *matrix multiplication*. More precisely, if our inputs are in a matrix `x` with a size of `batch_size` by `n_inputs`, and if we have grouped the weights of our neurons in a matrix `w` of size `n_neurons` by `n_inputs` (each neuron must have the same number of weights as it has inputs) and all the biases in a vector `b` of size `n_neurons`, then the output of this fully connected layer is:

```
    y = x @ w.t() + b
```

where `@` represents the matrix product and `w.t()` is the transpose matrix of `w`. The output `y` is then of size `batch_size` by `n_neurons`, and in position `(i,j)` we have (for the mathy folks out there):

$$y_{i,j} = \sum_{k=1}^{n} x_{i,k} w_{k,j} + b_j$$

Or in code:

```
    y[i,j] = sum([a * b for a,b in zip(x[i,:],w[j,:])]) + b[j]
```

The transpose is necessary because in the mathematical definition of the matrix product `m @ n`, the coefficient `(i,j)` is:

```
    sum([a * b for a,b in zip(m[i,:],n[:,j])])
```

So the very basic operation we need is a matrix multiplication, as it's what is hidden in the core of a neural net.

Matrix Multiplication from Scratch

Let's write a function that computes the matrix product of two tensors, before we allow ourselves to use the PyTorch version of it. We will only use the indexing in PyTorch tensors:

In [ ]:
```
import torch
from torch import tensor
```

We'll need three nested `for` loops: one for the row indices, one for the column indices, and one for the inner sum. `ac` and `ar` stand for number of columns of `a` and number of rows of `a`, respectively (the same convention is followed for `b`), and we make sure calculating the matrix product is possible by checking that `a` has as many columns as `b` has rows:

In [ ]:
```
def matmul(a,b):
    ar,ac = a.shape # n_rows * n_cols
    br,bc = b.shape
    assert ac==br
    c = torch.zeros(ar, bc)
    for i in range(ar):
        for j in range(bc):
            for k in range(ac): c[i,j] += a[i,k] * b[k,j]
    return c
```

To test this out, we'll pretend (using random matrices) that we're working with a small batch of 5 MNIST images, flattened into 28×28 vectors, with linear model to turn them into 10 activations:

In [ ]:
```
m1 = torch.randn(5,28*28)
m2 = torch.randn(784,10)
```

Let's time our function, using the Jupyter "magic" command `%time`:

In [ ]:
```
%time t1=matmul(m1, m2)
```

```
CPU times: user 1.15 s, sys: 4.09 ms, total: 1.15 s
Wall time: 1.15 s
```

And see how that compares to PyTorch's built-in `@`:

In [ ]:
```
%timeit -n 20 t2=m1@m2
```

```
14 µs ± 8.95 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

As we can see, in Python three nested loops is a very bad idea! Python is a slow language, and this isn't going to be very efficient. We see here that PyTorch is around 100,000 times faster than Python—and that's before we even start using the GPU!

Where does this difference come from? PyTorch didn't write its matrix multiplication in Python, but rather in C++ to make it fast. In general, whenever we do computations on tensors we will need to *vectorize* them so that we can take advantage of the speed of PyTorch, usually by using two techniques: elementwise arithmetic and broadcasting.

Elementwise Arithmetic

All the basic operators (`+` , `-` , `*` , `/` , `>` , `<` , `==`) can be applied elementwise. That means if we write `a+b` for two tensors `a` and `b` that have the same shape, we will get a tensor composed of the sums the elements of `a` and `b` :

In [ ]:
```
a = tensor([10., 6, -4])
b = tensor([2., 8, 7])
a + b
```

Out[ ]:
```
tensor([12., 14.,  3.])
```

The Booleans operators will return an array of Booleans:

In [ ]:
```
a < b
```

Out[ ]:
```
tensor([False,  True,  True])
```

If we want to know if every element of `a` is less than the corresponding element in `b` , or if two tensors are equal, we need to combine those elementwise operations with `torch.all` :

In [ ]:
```
(a < b).all(), (a==b).all()
```

Out[ ]:
```
(tensor(False), tensor(False))
```

Reduction operations like `all()` , `sum()` and `mean()` return tensors with only one element, called rank-0 tensors. If you want to convert this to a plain Python Boolean or number, you need to call `.item()` :

In [ ]:
```
(a + b).mean().item()
```

Out[ ]:
```
9.666666984558105
```

The elementwise operations work on tensors of any rank, as long as they have the same shape:

In [ ]:
```
m = tensor([[1., 2, 3], [4,5,6], [7,8,9]])
m*m
```

Out[ ]:
```
tensor([[ 1.,  4.,  9.],
        [16., 25., 36.],
        [49., 64., 81.]])
```

However you can't perform elementwise operations on tensors that don't have the same shape (unless they are broadcastable, as discussed in the next section):

In [ ]:
```
n = tensor([[1., 2, 3], [4,5,6]])
m*n
```

```
---------------------------------------------------------------------
--
RuntimeError                              Traceback (most recent call las
t)
<ipython-input-12-add73c4f74e0> in <module>
      1 n = tensor([[1., 2, 3], [4,5,6]])
----> 2 m*n

RuntimeError: The size of tensor a (3) must match the size of tensor b
(2) at non-singleton dimension 0
```

With elementwise arithmetic, we can remove one of our three nested loops: we can multiply the tensors that correspond to the `i`-th row of `a` and the `j`-th column of `b` before summing all the elements, which will speed things up because the inner loop will now be executed by PyTorch at C speed.

To access one column or row, we can simply write `a[i,:]` or `b[:,j]`. The `:` means take everything in that dimension. We could restrict this and take only a slice of that particular dimension by passing a range, like `1:5`, instead of just `:`. In that case, we would take the elements in columns or rows 1 to 4 (the second number is noninclusive).

One simplification is that we can always omit a trailing colon, so `a[i,:]` can be abbreviated to `a[i]`. With all of that in mind, we can write a new version of our matrix multiplication:

In [ ]:
```
def matmul(a,b):
    ar,ac = a.shape
    br,bc = b.shape
    assert ac==br
    c = torch.zeros(ar, bc)
    for i in range(ar):
        for j in range(bc): c[i,j] = (a[i] * b[:,j]).sum()
    return c
```

In [ ]:
```
%timeit -n 20 t3 = matmul(m1,m2)
```

```
1.7 ms ± 88.1 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

We're already ~700 times faster, just by removing that inner `for` loop! And that's just the beginning—with broadcasting we can remove another loop and get an even more important speed up.

Broadcasting

As we discussed in <>, broadcasting is a term introduced by the [NumPy library (https://docs.scipy.org/doc/)](https://docs.scipy.org/doc/) that describes how tensors of different ranks are treated during arithmetic operations. For instance, it's obvious there is no way to add a 3×3 matrix with a 4×5 matrix, but what if we want to add one scalar (which can be represented as a 1×1 tensor) with a matrix? Or a vector of size 3 with a 3×4 matrix? In both cases, we can find a way to make sense of this operation.

Broadcasting gives specific rules to codify when shapes are compatible when trying to do an elementwise operation, and how the tensor of the smaller shape is expanded to match the tensor of the bigger shape. It's essential to master those rules if you want to be able to write code that executes quickly. In this section, we'll expand our previous treatment of broadcasting to understand these rules.

Broadcasting with a scalar

Broadcasting with a scalar is the easiest type of broadcasting. When we have a tensor `a` and a scalar, we just imagine a tensor of the same shape as `a` filled with that scalar and perform the operation:

In [ ]:
```
a = tensor([10., 6, -4])
a > 0
```

Out[ ]:
```
tensor([ True,  True, False])
```

How are we able to do this comparison? `0` is being *broadcast* to have the same dimensions as `a`. Note that this is done without creating a tensor full of zeros in memory (that would be very inefficient).

This is very useful if you want to normalize your dataset by subtracting the mean (a scalar) from the entire data set (a matrix) and dividing by the standard deviation (another scalar):

In [ ]:
```
m = tensor([[1., 2, 3], [4,5,6], [7,8,9]])
(m - 5) / 2.73
```

Out[ ]:
```
tensor([[-1.4652, -1.0989, -0.7326],
        [-0.3663,  0.0000,  0.3663],
        [ 0.7326,  1.0989,  1.4652]])
```

What if have different means for each row of the matrix? in that case you will need to broadcast a vector to a matrix.

Broadcasting a vector to a matrix

We can broadcast a vector to a matrix as follows:

In [ ]:
```
c = tensor([10.,20,30])
m = tensor([[1., 2, 3], [4,5,6], [7,8,9]])
m.shape,c.shape
```

Out[ ]:
```
(torch.Size([3, 3]), torch.Size([3]))
```

In [ ]:
```
m + c
```

Out[ ]:
```
tensor([[11., 22., 33.],
        [14., 25., 36.],
        [17., 28., 39.]])
```

Here the elements of `c` are expanded to make three rows that match, making the operation possible. Again, PyTorch doesn't actually create three copies of `c` in memory. This is done by the `expand_as` method behind the scenes:

In [ ]:
```
c.expand_as(m)
```

Out[ ]:
```
tensor([[10., 20., 30.],
        [10., 20., 30.],
        [10., 20., 30.]])
```

If we look at the corresponding tensor, we can ask for its `storage` property (which shows the actual contents of the memory used for the tensor) to check there is no useless data stored:

In [ ]:
```
t = c.expand_as(m)
t.storage()
```

Out[ ]:
```
 10.0
 20.0
 30.0
[torch.FloatStorage of size 3]
```

Even though the tensor officially has nine elements, only three scalars are stored in memory. This is possible thanks to the clever trick of giving that dimension a *stride* of 0 (which means that when PyTorch looks for the next row by adding the stride, it doesn't move):

In [ ]:
```
t.stride(), t.shape
```

Out[ ]:
```
((0, 1), torch.Size([3, 3]))
```

Since `m` is of size 3×3, there are two ways to do broadcasting. The fact it was done on the last dimension is a convention that comes from the rules of broadcasting and has nothing to do with the way we ordered our tensors. If instead we do this, we get the same result:

In [ ]:
```
c + m
```

Out[ ]:
```
tensor([[11., 22., 33.],
        [14., 25., 36.],
        [17., 28., 39.]])
```

In fact, it's only possible to broadcast a vector of size `n` with a matrix of size `m` by `n`:

In [ ]:
```
c = tensor([10.,20,30])
m = tensor([[1., 2, 3], [4,5,6]])
c+m
```

Out[ ]:
```
tensor([[11., 22., 33.],
        [14., 25., 36.]])
```

This won't work:

In [ ]:
```
c = tensor([10.,20])
m = tensor([[1., 2, 3], [4,5,6]])
c+m
```

```
---------------------------------------------------------------------------
--
RuntimeError                              Traceback (most recent call las
t)
<ipython-input-25-64bbbad4d99c> in <module>
      1 c = tensor([10.,20])
      2 m = tensor([[1., 2, 3], [4,5,6]])
----> 3 c+m

RuntimeError: The size of tensor a (2) must match the size of tensor b
(3) at non-singleton dimension 1
```

If we want to broadcast in the other dimension, we have to change the shape of our vector to make it a 3×1 matrix. This is done with the `unsqueeze` method in PyTorch:

In [ ]:
```
c = tensor([10.,20,30])
m = tensor([[1., 2, 3], [4,5,6], [7,8,9]])
c = c.unsqueeze(1)
m.shape,c.shape
```

Out[ ]:
```
(torch.Size([3, 3]), torch.Size([3, 1]))
```

This time, `c` is expanded on the column side:

In [ ]:
```
c+m
```

Out[ ]:
```
tensor([[11., 12., 13.],
        [24., 25., 26.],
        [37., 38., 39.]])
```

Like before, only three scalars are stored in memory:

In [ ]:
```
t = c.expand_as(m)
t.storage()
```

Out[ ]:
```
 10.0
 20.0
 30.0
[torch.FloatStorage of size 3]
```

And the expanded tensor has the right shape because the column dimension has a stride of 0:

In [ ]:
```
t.stride(), t.shape
```

Out[ ]:
```
((1, 0), torch.Size([3, 3]))
```

With broadcasting, by default if we need to add dimensions, they are added at the beginning. When we were broadcasting before, Pytorch was doing `c.unsqueeze(0)` behind the scenes:

In [ ]:
```
c = tensor([10.,20,30])
c.shape, c.unsqueeze(0).shape,c.unsqueeze(1).shape
```

Out[ ]:
```
(torch.Size([3]), torch.Size([1, 3]), torch.Size([3, 1]))
```

The `unsqueeze` command can be replaced by `None` indexing:

In [ ]:
```
c.shape, c[None,:].shape,c[:,None].shape
```

Out[ ]:
```
(torch.Size([3]), torch.Size([1, 3]), torch.Size([3, 1]))
```

You can always omit trailing colons, and `...` means all preceding dimensions:

In [ ]:
```
c[None].shape,c[...,None].shape
```

Out[ ]:
```
(torch.Size([1, 3]), torch.Size([3, 1]))
```

With this, we can remove another `for` loop in our matrix multiplication function. Now, instead of multiplying `a[i]` with `b[:,j]`, we can multiply `a[i]` with the whole matrix `b` using broadcasting, then sum the results:

In [ ]:
```
def matmul(a,b):
    ar,ac = a.shape
    br,bc = b.shape
    assert ac==br
    c = torch.zeros(ar, bc)
    for i in range(ar):
#       c[i,j] = (a[i,:]          * b[:,j]).sum() # previous
        c[i]   = (a[i  ].unsqueeze(-1) * b).sum(dim=0)
    return c
```

In [ ]:
```
%timeit -n 20 t4 = matmul(m1,m2)
```

```
357 µs ± 7.2 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

We're now 3,700 times faster than our first implementation! Before we move on, let's discuss the rules of broadcasting in a little more detail.

Broadcasting rules

When operating on two tensors, PyTorch compares their shapes elementwise. It starts with the *trailing dimensions* and works its way backward, adding 1 when it meets empty dimensions. Two dimensions are *compatible* when one of the following is true:

- They are equal.
- One of them is 1, in which case that dimension is broadcast to make it the same as the other.

Arrays do not need to have the same number of dimensions. For example, if you have a 256×256×3 array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with three values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:

```
Image  (3d tensor): 256 x 256 x 3
Scale  (1d tensor):  (1)   (1)  3
Result (3d tensor): 256 x 256 x 3
```

However, a 2D tensor of size 256×256 isn't compatible with our image:

```
Image  (3d tensor): 256 x 256 x   3
Scale  (2d tensor):  (1)  256 x 256
Error
```

In our earlier examples we had with a 3×3 matrix and a vector of size 3, broadcasting was done on the rows:

```
Matrix (2d tensor):   3 x 3
Vector (1d tensor): (1)   3
Result (2d tensor):   3 x 3
```

As an exercise, try to determine what dimensions to add (and where) when you need to normalize a batch of images of size `64 x 3 x 256 x 256` with vectors of three elements (one for the mean and one for the standard deviation).

Another useful way of simplifying tensor manipulations is the use of Einstein summations convention.

Einstein Summation

Before using the PyTorch operation `@` or `torch.matmul`, there is one last way we can implement matrix multiplication: Einstein summation (`einsum`). This is a compact representation for combining products

and sums in a general way. We write an equation like this:

```
ik,kj -> ij
```

The lefthand side represents the operands dimensions, separated by commas. Here we have two tensors that each have two dimensions (i,k and k,j). The righthand side represents the result dimensions, so here we have a tensor with two dimensions i,j .

The rules of Einstein summation notation are as follows:

1. Repeated indices on the left side are implicitly summed over if they are not on the right side.
2. Each index can appear at most twice on the left side.
3. The unrepeated indices on the left side must appear on the right side.

So in our example, since k is repeated, we sum over that index. In the end the formula represents the matrix obtained when we put in (i,j) the sum of all the coefficients (i,k) in the first tensor multiplied by the coefficients (k,j) in the second tensor... which is the matrix product! Here is how we can code this in PyTorch:

In [ ]:
```
def matmul(a,b): return torch.einsum('ik,kj->ij', a, b)
```

Einstein summation is a very practical way of expressing operations involving indexing and sum of products. Note that you can have just one member on the lefthand side. For instance, this:

```
torch.einsum('ij->ji', a)
```

returns the transpose of the matrix a . You can also have three or more members. This:

```
torch.einsum('bi,ij,bj->b', a, b, c)
```

will return a vector of size b where the k -th coordinate is the sum of a[k,i] b[i,j] c[k,j] . This notation is particularly convenient when you have more dimensions because of batches. For example, if you have two batches of matrices and want to compute the matrix product per batch, you would could this:

```
torch.einsum('bik,bkj->bij', a, b)
```

Let's go back to our new matmul implementation using einsum and look at its speed:

In [ ]:
```
%timeit -n 20 t5 = matmul(m1,m2)
```

```
68.7 µs ± 4.06 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

As you can see, not only is it practical, but it's *very* fast. `einsum` is often the fastest way to do custom operations in PyTorch, without diving into C++ and CUDA. (But it's generally not as fast as carefully optimized CUDA code, as you see from the results in "Matrix Multiplication from Scratch".)

Now that we know how to implement a matrix multiplication from scratch, we are ready to build our neural net—specifically its forward and backward passes—using just matrix multiplications.

The Forward and Backward Passes

As we saw in <>, to train a model, we will need to compute all the gradients of a given loss with respect to its parameters, which is known as the *backward pass*. The *forward pass* is where we compute the output of the model on a given input, based on the matrix products. As we define our first neural net, we will also delve into the problem of properly initializing the weights, which is crucial for making training start properly.

Defining and Initializing a Layer

We will take the example of a two-layer neural net first. As we've seen, one layer can be expressed as `y = x @ w + b`, with `x` our inputs, `y` our outputs, `w` the weights of the layer (which is of size number of inputs by number of neurons if we don't transpose like before), and `b` is the bias vector:

In [ ]:
```
def lin(x, w, b): return x @ w + b
```

We can stack the second layer on top of the first, but since mathematically the composition of two linear operations is another linear operation, this only makes sense if we put something nonlinear in the middle, called an activation function. As mentioned at the beginning of the chapter, in deep learning applications the activation function most commonly used is a ReLU, which returns the maximum of `x` and `0`.

We won't actually train our model in this chapter, so we'll use random tensors for our inputs and targets. Let's say our inputs are 200 vectors of size 100, which we group into one batch, and our targets are 200 random floats:

In [ ]:
```
x = torch.randn(200, 100)
y = torch.randn(200)
```

For our two-layer model we will need two weight matrices and two bias vectors. Let's say we have a hidden size of 50 and the output size is 1 (for one of our inputs, the corresponding output is one float in this toy example). We initialize the weights randomly and the bias at zero:

In [ ]:
```
w1 = torch.randn(100,50)
b1 = torch.zeros(50)
w2 = torch.randn(50,1)
b2 = torch.zeros(1)
```

Then the result of our first layer is simply:

In [ ]:
```
l1 = lin(x, w1, b1)
l1.shape
```

Out[ ]:
```
torch.Size([200, 50])
```

Note that this formula works with our batch of inputs, and returns a batch of hidden state: `l1` is a matrix of size 200 (our batch size) by 50 (our hidden size).

There is a problem with the way our model was initialized, however. To understand it, we need to look at the mean and standard deviation (std) of `l1`:

In [ ]:
```
l1.mean(), l1.std()
```

Out[ ]:
```
(tensor(0.0019), tensor(10.1058))
```

The mean is close to zero, which is understandable since both our input and weight matrices have means close to zero. But the standard deviation, which represents how far away our activations go from the mean, went from 1 to 10. This is a really big problem because that's with just one layer. Modern neural nets can have hundred of layers, so if each of them multiplies the scale of our activations by 10, by the end of the last layer we won't have numbers representable by a computer.

Indeed, if we make just 50 multiplications between `x` and random matrices of size 100×100, we'll have:

In [ ]:
```
x = torch.randn(200, 100)
for i in range(50): x = x @ torch.randn(100,100)
x[0:5,0:5]
```

Out[ ]:
```
tensor([[nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan],
        [nan, nan, nan, nan, nan]])
```

The result is nan s everywhere. So maybe the scale of our matrix was too big, and we need to have smaller weights? But if we use too small weights, we will have the opposite problem—the scale of our activations will go from 1 to 0.1, and after 50 layers we'll be left with zeros everywhere:

In [ ]:
```
x = torch.randn(200, 100)
for i in range(50): x = x @ (torch.randn(100,100) * 0.01)
x[0:5,0:5]
```

Out[ ]:
```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

So we have to scale our weight matrices exactly right so that the standard deviation of our activations stays at 1. We can compute the exact value to use mathematically, as illustrated by Xavier Glorot and Yoshua Bengio in "Understanding the Difficulty of Training Deep Feedforward Neural Networks" (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf). The right scale for a given layer is $1/\sqrt{n_{in}}$, where n_{in} represents the number of inputs.

In our case, if we have 100 inputs, we should scale our weight matrices by 0.1:

In [ ]:
```
x = torch.randn(200, 100)
for i in range(50): x = x @ (torch.randn(100,100) * 0.1)
x[0:5,0:5]
```

Out[ ]:
```
tensor([[ 0.7554,  0.6167, -0.1757, -1.5662,  0.5644],
        [-0.1987,  0.6292,  0.3283, -1.1538,  0.5416],
        [ 0.6106,  0.2556, -0.0618, -0.9463,  0.4445],
        [ 0.4484,  0.7144,  0.1164, -0.8626,  0.4413],
        [ 0.3463,  0.5930,  0.3375, -0.9486,  0.5643]])
```

Finally some numbers that are neither zeros nor nan s! Notice how stable the scale of our activations is, even after those 50 fake layers:

In [ ]:
```
x.std()
```

Out[ ]:
```
tensor(0.7042)
```

If you play a little bit with the value for scale you'll notice that even a slight variation from 0.1 will get you either to very small or very large numbers, so initializing the weights properly is extremely important.

Let's go back to our neural net. Since we messed a bit with our inputs, we need to redefine them:

In [ ]:
```
x = torch.randn(200, 100)
y = torch.randn(200)
```

And for our weights, we'll use the right scale, which is known as *Xavier initialization* (or *Glorot initialization*):

In [ ]:
```
from math import sqrt
w1 = torch.randn(100,50) / sqrt(100)
b1 = torch.zeros(50)
w2 = torch.randn(50,1) / sqrt(50)
b2 = torch.zeros(1)
```

Now if we compute the result of the first layer, we can check that the mean and standard deviation are under control:

In [ ]:
```
l1 = lin(x, w1, b1)
l1.mean(),l1.std()
```

Out[ ]:
```
(tensor(-0.0050), tensor(1.0000))
```

Very good. Now we need to go through a ReLU, so let's define one. A ReLU removes the negatives and replaces them with zeros, which is another way of saying it clamps our tensor at zero:

In [ ]:
```
def relu(x): return x.clamp_min(0.)
```

We pass our activations through this:

In [ ]:
```
l2 = relu(l1)
l2.mean(),l2.std()
```

Out[ ]:
```
(tensor(0.3961), tensor(0.5783))
```

And we're back to square one: the mean of our activations has gone to 0.4 (which is understandable since we removed the negatives) and the std went down to 0.58. So like before, after a few layers we will probably wind up with zeros:

In [ ]:
```
x = torch.randn(200, 100)
for i in range(50): x = relu(x @ (torch.randn(100,100) * 0.1))
x[0:5,0:5]
```

Out[ ]:
```
tensor([[0.0000e+00, 1.9689e-08, 4.2820e-08, 0.0000e+00, 0.0000e+00],
        [0.0000e+00, 1.6701e-08, 4.3501e-08, 0.0000e+00, 0.0000e+00],
        [0.0000e+00, 1.0976e-08, 3.0411e-08, 0.0000e+00, 0.0000e+00],
        [0.0000e+00, 1.8457e-08, 4.9469e-08, 0.0000e+00, 0.0000e+00],
        [0.0000e+00, 1.9949e-08, 4.1643e-08, 0.0000e+00, 0.0000e+00]])
```

This means our initialization wasn't right. Why? At the time Glorot and Bengio wrote their article, the popular activation in a neural net was the hyperbolic tangent (tanh, which is the one they used), and that initialization doesn't account for our ReLU. Fortunately, someone else has done the math for us and computed the right scale for us to use. In "Delving Deep into Rectifiers: Surpassing Human-Level Performance" (https://arxiv.org/abs/1502.01852) (which we've seen before—it's the article that introduced the ResNet), Kaiming He et al. show that we should use the following scale instead: $\sqrt{2 / n_{in}}$, where n_{in} is the number of inputs of our model. Let's see what this gives us:

In [ ]:
```
x = torch.randn(200, 100)
for i in range(50): x = relu(x @ (torch.randn(100,100) * sqrt(2/100)))
x[0:5,0:5]
```

Out[ ]:
```
tensor([[0.2871, 0.0000, 0.0000, 0.0000, 0.0026],
        [0.4546, 0.0000, 0.0000, 0.0000, 0.0015],
        [0.6178, 0.0000, 0.0000, 0.0180, 0.0079],
        [0.3333, 0.0000, 0.0000, 0.0545, 0.0000],
        [0.1940, 0.0000, 0.0000, 0.0000, 0.0096]])
```

That's better: our numbers aren't all zeroed this time. So let's go back to the definition of our neural net and use this initialization (which is named *Kaiming initialization* or *He initialization*):

In [ ]:
```
x = torch.randn(200, 100)
y = torch.randn(200)
```

In [ ]:
```
w1 = torch.randn(100,50) * sqrt(2 / 100)
b1 = torch.zeros(50)
w2 = torch.randn(50,1) * sqrt(2 / 50)
b2 = torch.zeros(1)
```

Let's look at the scale of our activations after going through the first linear layer and ReLU:

In [ ]:
```
l1 = lin(x, w1, b1)
l2 = relu(l1)
l2.mean(), l2.std()
```

Out[ ]:
```
(tensor(0.5661), tensor(0.8339))
```

Much better! Now that our weights are properly initialized, we can define our whole model:

In [ ]:
```
def model(x):
    l1 = lin(x, w1, b1)
    l2 = relu(l1)
    l3 = lin(l2, w2, b2)
    return l3
```

This is the forward pass. Now all that's left to do is to compare our output to the labels we have (random numbers, in this example) with a loss function. In this case, we will use the mean squared error. (It's a toy problem, and this is the easiest loss function to use for what is next, computing the gradients.)

The only subtlety is that our outputs and targets don't have exactly the same shape—after going though the model, we get an output like this:

In [ ]:
```
out = model(x)
out.shape
```

Out[ ]:
```
torch.Size([200, 1])
```

To get rid of this trailing 1 dimension, we use the squeeze function:

In [ ]:
```
def mse(output, targ): return (output.squeeze(-1) - targ).pow(2).mean()
```

And now we are ready to compute our loss:

In [ ]:
```
loss = mse(out, y)
```

That's all for the forward pass—let's now look at the gradients.

Gradients and the Backward Pass

We've seen that PyTorch computes all the gradients we need with a magic call to `loss.backward`, but let's explore what's happening behind the scenes.

Now comes the part where we need to compute the gradients of the loss with respect to all the weights of our model, so all the floats in `w1`, `b1`, `w2`, and `b2`. For this, we will need a bit of math—specifically the *chain rule*. This is the rule of calculus that guides how we can compute the derivative of a composed function:

$$(g \circ f)'(x) = g'(f(x))f'(x)$$

> j: I find this notation very hard to wrap my head around, so instead I like to think of it as: if `y = g(u)` and `u=f(x)`; then `dy/dx = dy/du * du/dx`. The two notations mean the same thing, so use whatever works for you.

Our loss is a big composition of different functions: mean squared error (which is in turn the composition of a mean and a power of two), the second linear layer, a ReLU and the first linear layer. For instance, if we want the gradients of the loss with respect to `b2` and our loss is defined by:

```
loss = mse(out,y) = mse(lin(l2, w2, b2), y)
```

The chain rule tells us that we have:

$$\frac{\mathrm{d}loss}{\mathrm{d}b_2} = \frac{\mathrm{d}loss}{\mathrm{d}out} \times \frac{\mathrm{d}out}{\mathrm{d}b_2} = \frac{\mathrm{d}}{\mathrm{d}out}mse(out, y) \times \frac{\mathrm{d}}{\mathrm{d}b_2}lin(l_2, w_2, b_2)$$

To compute the gradients of the loss with respect to b_2, we first need the gradients of the loss with respect to our output out. It's the same if we want the gradients of the loss with respect to w_2. Then, to get the gradients of the loss with respect to b_1 or w_1, we will need the gradients of the loss with respect to l_1, which in turn requires the gradients of the loss with respect to l_2, which will need the gradients of the loss with respect to out.

So to compute all the gradients we need for the update, we need to begin from the output of the model and work our way *backward*, one layer after the other—which is why this step is known as *backpropagation*. We can automate it by having each function we implemented (relu , mse , lin) provide its backward step: that is, how to derive the gradients of the loss with respect to the input(s) from the gradients of the loss with respect to the output.

Here we populate those gradients in an attribute of each tensor, a bit like PyTorch does with .grad .

The first are the gradients of the loss with respect to the output of our model (which is the input of the loss function). We undo the squeeze we did in mse , then we use the formula that gives us the derivative of x^2: $2x$. The derivative of the mean is just $1/n$ where n is the number of elements in our input:

In [ ]:
```
def mse_grad(inp, targ):
    # grad of loss with respect to output of previous layer
    inp.g = 2. * (inp.squeeze() - targ).unsqueeze(-1) / inp.shape[0]
```

For the gradients of the ReLU and our linear layer, we use the gradients of the loss with respect to the output (in out.g) and apply the chain rule to compute the gradients of the loss with respect to the input (in inp.g). The chain rule tells us that inp.g = relu'(inp) * out.g . The derivative of relu is either 0 (when inputs are negative) or 1 (when inputs are positive), so this gives us:

In [ ]:
```
def relu_grad(inp, out):
    # grad of relu with respect to input activations
    inp.g = (inp>0).float() * out.g
```

The scheme is the same to compute the gradients of the loss with respect to the inputs, weights, and bias in the linear layer:

In [ ]:
```
def lin_grad(inp, out, w, b):
    # grad of matmul with respect to input
    inp.g = out.g @ w.t()
    w.g = inp.t() @ out.g
    b.g = out.g.sum(0)
```

We won't linger on the mathematical formulas that define them since they're not important for our purposes, but do check out Khan Academy's excellent calculus lessons if you're interested in this topic.

Sidebar: SymPy

SymPy is a library for symbolic computation that is extremely useful library when working with calculus. Per the documentation (https://docs.sympy.org/latest/tutorial/intro.html):

> : Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

To do symbolic computation, we first define a *symbol*, and then do a computation, like so:

In [ ]:
```
from sympy import symbols,diff
sx,sy = symbols('sx sy')
diff(sx**2, sx)
```

Out[ ]: $2sx$

Here, SymPy has taken the derivative of `x**2` for us! It can take the derivative of complicated compound expressions, simplify and factor equations, and much more. There's really not much reason for anyone to do calculus manually nowadays—for calculating gradients, PyTorch does it for us, and for showing the equations, SymPy does it for us!

End sidebar

Once we have have defined those functions, we can use them to write the backward pass. Since each gradient is automatically populated in the right tensor, we don't need to store the results of those `_grad` functions anywhere—we just need to execute them in the reverse order of the forward pass, to make sure that in each function `out.g` exists:

In [ ]:
```
def forward_and_backward(inp, targ):
    # forward pass:
    l1 = inp @ w1 + b1
    l2 = relu(l1)
    out = l2 @ w2 + b2
    # we don't actually need the loss in backward!
    loss = mse(out, targ)

    # backward pass:
    mse_grad(out, targ)
    lin_grad(l2, out, w2, b2)
    relu_grad(l1, l2)
    lin_grad(inp, l1, w1, b1)
```

And now we can access the gradients of our model parameters in `w1.g`, `b1.g`, `w2.g`, and `b2.g`.

We have successfully defined our model—now let's make it a bit more like a PyTorch module.

Refactoring the Model

The three functions we used have two associated functions: a forward pass and a backward pass. Instead of writing them separately, we can create a class to wrap them together. That class can also store the inputs and outputs for the backward pass. This way, we will just have to call `backward`:

In [ ]:
```
class Relu():
    def __call__(self, inp):
        self.inp = inp
        self.out = inp.clamp_min(0.)
        return self.out

    def backward(self): self.inp.g = (self.inp>0).float() * self.out.g
```

`__call__` is a magic name in Python that will make our class callable. This is what will be executed when we type `y = Relu()(x)`. We can do the same for our linear layer and the MSE loss:

In [ ]:
```
class Lin():
    def __init__(self, w, b): self.w,self.b = w,b

    def __call__(self, inp):
        self.inp = inp
        self.out = inp@self.w + self.b
        return self.out

    def backward(self):
        self.inp.g = self.out.g @ self.w.t()
        self.w.g = self.inp.t() @ self.out.g
        self.b.g = self.out.g.sum(0)
```

In [ ]:
```
class Mse():
    def __call__(self, inp, targ):
        self.inp = inp
        self.targ = targ
        self.out = (inp.squeeze() - targ).pow(2).mean()
        return self.out

    def backward(self):
        x = (self.inp.squeeze()-self.targ).unsqueeze(-1)
        self.inp.g = 2.*x/self.targ.shape[0]
```

Then we can put everything in a model that we initiate with our tensors `w1`, `b1`, `w2`, `b2`:

In [ ]:
```
class Model():
    def __init__(self, w1, b1, w2, b2):
        self.layers = [Lin(w1,b1), Relu(), Lin(w2,b2)]
        self.loss = Mse()

    def __call__(self, x, targ):
        for l in self.layers: x = l(x)
        return self.loss(x, targ)

    def backward(self):
        self.loss.backward()
        for l in reversed(self.layers): l.backward()
```

What is really nice about this refactoring and registering things as layers of our model is that the forward and backward passes are now really easy to write. If we want to instantiate our model, we just need to write:

In [ ]:
```
model = Model(w1, b1, w2, b2)
```

The forward pass can then be executed with:

In [ ]:
```
loss = model(x, y)
```

And the backward pass with:

In [ ]:
```
model.backward()
```

Going to PyTorch

The `Lin`, `Mse` and `Relu` classes we wrote have a lot in common, so we could make them all inherit from the same base class:

In [ ]:
```
class LayerFunction():
    def __call__(self, *args):
        self.args = args
        self.out = self.forward(*args)
        return self.out

    def forward(self):  raise Exception('not implemented')
    def bwd(self):      raise Exception('not implemented')
    def backward(self): self.bwd(self.out, *self.args)
```

Then we just need to implement `forward` and `bwd` in each of our subclasses:

In [ ]:
```
class Relu(LayerFunction):
    def forward(self, inp): return inp.clamp_min(0.)
    def bwd(self, out, inp): inp.g = (inp>0).float() * out.g
```

In [ ]:
```
class Lin(LayerFunction):
    def __init__(self, w, b): self.w,self.b = w,b

    def forward(self, inp): return inp@self.w + self.b

    def bwd(self, out, inp):
        inp.g = out.g @ self.w.t()
        self.w.g = inp.t() @ self.out.g
        self.b.g = out.g.sum(0)
```

In [ ]:
```
class Mse(LayerFunction):
    def forward (self, inp, targ): return (inp.squeeze() - targ).pow(2)
    def bwd(self, out, inp, targ):
        inp.g = 2*(inp.squeeze()-targ).unsqueeze(-1) / targ.shape[0]
```

The rest of our model can be the same as before. This is getting closer and closer to what PyTorch does. Each basic function we need to differentiate is written as a `torch.autograd.Function` object that has a `forward` and a `backward` method. PyTorch will then keep trace

of any computation we do to be able to properly run the backward pass, unless we set the `requires_grad` attribute of our tensors to `False`.

Writing one of these is (almost) as easy as writing our original classes. The difference is that we choose what to save and what to put in a context variable (so that we make sure we don't save anything we don't need), and we return the gradients in the `backward` pass. It's very rare to have to write your own `Function` but if you

In [ ]:
```
from torch.autograd import Function

class MyRelu(Function):
    @staticmethod
    def forward(ctx, i):
        result = i.clamp_min(0.)
        ctx.save_for_backward(i)
        return result

    @staticmethod
    def backward(ctx, grad_output):
        i, = ctx.saved_tensors
        return grad_output * (i>0).float()
```

The structure used to build a more complex model that takes advantage of those `Function`s is a `torch.nn.Module`. This is the base structure for all models, and all the neural nets you have seen up until now inherited from that class. It mostly helps to register all the trainable parameters, which as we've seen can be used in the training loop.

To implement an `nn.Module` you just need to:

- Make sure the superclass `__init__` is called first when you initialize it.
- Define any parameters of the model as attributes with `nn.Parameter`.
- Define a `forward` function that returns the output of your model.

As an example, here is the linear layer from scratch:

In [ ]:
```
import torch.nn as nn

class LinearLayer(nn.Module):
    def __init__(self, n_in, n_out):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(n_out, n_in) * sqrt(2/n_
        self.bias = nn.Parameter(torch.zeros(n_out))

    def forward(self, x): return x @ self.weight.t() + self.bias
```

As you see, this class automatically keeps track of what parameters have been defined:

In [ ]:
```
lin = LinearLayer(10,2)
p1,p2 = lin.parameters()
p1.shape,p2.shape
```

Out[ ]:
```
(torch.Size([2, 10]), torch.Size([2]))
```

It is thanks to this feature of `nn.Module` that we can just say `opt.step()` and have an optimizer loop through the parameters and update each one.

Note that in PyTorch, the weights are stored as an `n_out x n_in` matrix, which is why we have the transpose in the forward pass.

By using the linear layer from PyTorch (which uses the Kaiming initialization as well), the model we have been building up during this chapter can be written like this:

In [ ]:
```
class Model(nn.Module):
    def __init__(self, n_in, nh, n_out):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(n_in,nh), nn.ReLU(), nn.Linear(nh,n_out))
        self.loss = mse

    def forward(self, x, targ): return self.loss(self.layers(x).squeeze
```

fastai provides its own variant of `Module` that is identical to `nn.Module`, but doesn't require you to call `super().__init__()` (it does that for you automatically):

In [ ]:
```
class Model(Module):
    def __init__(self, n_in, nh, n_out):
        self.layers = nn.Sequential(
            nn.Linear(n_in,nh), nn.ReLU(), nn.Linear(nh,n_out))
        self.loss = mse

    def forward(self, x, targ): return self.loss(self.layers(x).squeeze
```

In the last chapter, we will start from such a model and see how to build a training loop from scratch and refactor it to what we've been using in previous chapters.

Conclusion

In this chapter we explored the foundations of deep learning, beginning with matrix multiplication and moving on to implementing the forward and backward passes of a neural net from scratch. We then refactored our code to show how PyTorch works beneath the hood.

Here are a few things to remember:

- A neural net is basically a bunch of matrix multiplications with nonlinearities in between.
- Python is slow, so to write fast code we have to vectorize it and take advantage of techniques such as elementwise arithmetic and broadcasting.
- Two tensors are broadcastable if the dimensions starting from the end and going backward match (if they are the same, or one of them is 1). To make tensors broadcastable, we may need to add dimensions of size 1 with `unsqueeze` or a `None` index.
- Properly initializing a neural net is crucial to get training started. Kaiming initialization should be used when we have ReLU nonlinearities.
- The backward pass is the chain rule applied multiple times, computing the gradients from the output of our model and going back, one layer at a time.
- When subclassing `nn.Module` (if not using fastai's `Module`) we have to call the superclass `__init__` method in our `__init__` method and we have to define a `forward` function that takes an input and returns the desired result.

Questionnaire

1. Write the Python code to implement a single neuron.
2. Write the Python code to implement ReLU.

3. Write the Python code for a dense layer in terms of matrix multiplication.
4. Write the Python code for a dense layer in plain Python (that is, with list comprehensions and functionality built into Python).
5. What is the "hidden size" of a layer?
6. What does the `t` method do in PyTorch?
7. Why is matrix multiplication written in plain Python very slow?
8. In `matmul`, why is `ac==br`?
9. In Jupyter Notebook, how do you measure the time taken for a single cell to execute?
10. What is "elementwise arithmetic"?
11. Write the PyTorch code to test whether every element of `a` is greater than the corresponding element of `b`.
12. What is a rank-0 tensor? How do you convert it to a plain Python data type?
13. What does this return, and why? `tensor([1,2]) + tensor([1])`
14. What does this return, and why? `tensor([1,2]) + tensor([1,2,3])`
15. How does elementwise arithmetic help us speed up `matmul`?
16. What are the broadcasting rules?
17. What is `expand_as`? Show an example of how it can be used to match the results of broadcasting.
18. How does `unsqueeze` help us to solve certain broadcasting problems?
19. How can we use indexing to do the same operation as `unsqueeze`?
20. How do we show the actual contents of the memory used for a tensor?
21. When adding a vector of size 3 to a matrix of size 3×3, are the elements of the vector added to each row or each column of the matrix? (Be sure to check your answer by running this code in a notebook.)
22. Do broadcasting and `expand_as` result in increased memory use? Why or why not?
23. Implement `matmul` using Einstein summation.
24. What does a repeated index letter represent on the left-hand side of einsum?
25. What are the three rules of Einstein summation notation? Why?
26. What are the forward pass and backward pass of a neural network?
27. Why do we need to store some of the activations calculated for intermediate layers in the forward pass?
28. What is the downside of having activations with a standard deviation too far away from 1?
29. How can weight initialization help avoid this problem?
30. What is the formula to initialize weights such that we get a standard deviation of 1 for a plain linear layer, and for a linear layer followed by ReLU?
31. Why do we sometimes have to use the `squeeze` method in loss functions?
32. What does the argument to the `squeeze` method do? Why might it be important to include this argument, even though PyTorch does not

require it?
33. What is the "chain rule"? Show the equation in either of the two forms presented in this chapter.
34. Show how to calculate the gradients of `mse(lin(l2, w2, b2), y)` using the chain rule.
35. What is the gradient of ReLU? Show it in math or code. (You shouldn't need to commit this to memory—try to figure it using your knowledge of the shape of the function.)
36. In what order do we need to call the `*_grad` functions in the backward pass? Why?
37. What is `__call__`?
38. What methods must we implement when writing a `torch.autograd.Function`?
39. Write `nn.Linear` from scratch, and test it works.

Further Research

1. Implement ReLU as a `torch.autograd.Function` and train a model with it.
2. If you are mathematically inclined, find out what the gradients of a linear layer are in mathematical notation. Map that to the implementation we saw in this chapter.
3. Learn about the `unfold` method in PyTorch, and use it along with matrix multiplication to implement your own 2D convolution function. Then train a CNN that uses it.
4. Implement everything in this chapter using NumPy instead of PyTorch.

In [ ]: