1)How does unsqueeze help us to solve certain broadcasting problems ?

Ans : 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:

a = tensor([10., 6, -4])

a > 0

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):

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

(m - 5) / 2.73

tensor([[-1.4652, -1.0989, -0.7326],

[-0.3663, 0.0000, 0.3663],

[ 0.7326, 1.0989, 1.4652]])

[4,5,6], [7,8,9]])

m.shape,c.shape

(torch.Size([3, 3]), torch.Size([3]))

m + c

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:

c.expand\_as(m)

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:

t = c.expand\_as(m)

t.storage()

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):

t.stride(), t.shape

((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:

c + m

tensor([[11., 22., 33.],

[14., 25., 36.]])

This won't work:

c = tensor([10.,20])

m = tensor([[1., 2, 3], [4,5,6]])

c+m

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:

c = tensor([10.,20,30])

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

c = c.unsqueeze(1)

m.shape,c.shape

(torch.Size([3, 3]), torch.Size([3, 1])

This time, c is expanded on the column side:

c+m

tensor([[11., 12., 13.],

[24., 25., 26.],

[37., 38., 39.]])

Like before, only three scalars are stored in memory:

t = c.expand\_as(m)

t.storage()

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:

t.stride(), t.shape

((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:

c = tensor([10.,20,30])

c.shape, c.unsqueeze(0).shape,c.unsqueeze(1).shape

(torch.Size([3]), torch.Size([1, 3]), torch.Size([3, 1]))

The unsqueeze command can be replaced by None indexing:

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

(torch.Size([3]), torch.Size([1, 3]), torch.Size([3, 1]))

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

c[None].shape,c[...,None].shape

(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:

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

%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.

2)How can we use indexing to do the same operation an unsqueeze.?

Ans ::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:

c = tensor([10.,20,30])

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

c = c.unsqueeze(1)

m.shape,c.shape

(torch.Size([3, 3]), torch.Size([3, 1]))

This time, c is expanded on the column side:

c+m

tensor([[11., 12., 13.],

[24., 25., 26.],

[37., 38., 39.]])

Like before, only three scalars are stored in memory:

t = c.expand\_as(m)

t.storage()

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:

t.stride(), t.shape

((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:

c = tensor([10.,20,30])

c.shape, c.unsqueeze(0).shape,c.unsqueeze(1).shape

(torch.Size([3]), torch.Size([1, 3]), torch.Size([3, 1]))

The unsqueeze command can be replaced by None indexing:

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

(torch.Size([3]), torch.Size([1, 3]), torch.Size([3, 1]))

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

c[None].shape,c[...,None].shape

(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:

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

%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.

3)how do we show the actual contents of the memory used for tensor ?

Ans : The commonly used way to store such data is in a single array that is laid out as a single, contiguous block within memory. More concretely, a 3x3x3 tensor would be stored simply as a single array of 27 values, one after the other.

4)when adding a vector of size 3 to a matrix size 3x3 are the elements of the vector added to each row or each column of matrix.?

Ans :,We can broadcast a vector to a matrix as follows:

c = tensor([10.,20,30])

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

m.shape,c.shape

(torch.Size([3, 3]), torch.Size([3]))

m + c

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:

c.expand\_as(m)

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:

t = c.expand\_as(m)

t.storage()

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):

t.stride(), t.shape

((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:

c + m

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:

c = tensor([10.,20,30])

m = tensor([[1., 2, 3], [4,5,6]])

c+m

tensor([[11., 22., 33.],

[14., 25., 36.]])

This won't work:

c = tensor([10.,20])

m = tensor([[1., 2, 3], [4,5,6]])

c+m

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:

c = tensor([10.,20,30])

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

c = c.unsqueeze(1)

m.shape,c.shape

(torch.Size([3, 3]), torch.Size([3, 1]))

This time, c is expanded on the column side:

c+m

tensor([[11., 12., 13.],

[24., 25., 26.],

[37., 38., 39.]])

Like before, only three scalars are stored in memory:

t = c.expand\_as(m)

t.storage()

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:

t.stride(), t.shape

((1, 0), torch.Size([3, 3])).

5)Do Broadcasting and expand\_as result in increased memory use ?why or why not ?

Ans : 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:

c.expand\_as(m)

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:

t = c.expand\_as(m)

t.storage()

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):

t.stride(), t.shape

((0, 1), torch.Size([3, 3])

6)Implement matmul using Einstein Summation.

Ans :,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 left hand side represents the operands dimensions, separated by commas. Here we have two tensors that each have two dimensions (i,k and k,j). The right hand 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:

Repeated indices on the left side are implicitly summed over if they are not on the right side.

Each index can appear at most twice on the left side.

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:

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 left hand 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:

%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.

7)what does a repeated index letter represent on the left hand side of einsum ?

ans : 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:

Repeated indices on the left side are implicitly summed over if they are not on the right side.

Each index can appear at most twice on the left side.

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:

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:

%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

8)what are the three rules of einstein summation notation?why ?

Ans :1. Each index can appear at most twice in any term.2. Repeated indices are implicitly summed over. 3.Each term must contain identical non-repeated indices.

9)what are the forward pass and backward pass of a neural network.?

Ans: Backward and forward pass makes together one "iteration". During one iteration, you usually pass a subset of the data set, which is called "mini-batch" or "batch" , "Epoch" means passing the entire data set in batches.

10)why do we need to store some of the activation calculated for intermediate layers in the forward pass ?

Ans :Forward propagation refers to storage and calculation of input data which is fed in forward direction through the network to generate an output. Hidden layers in neural network accepts the data from the input layer, process it on the basis of activation function and pass it to the output layer or the successive layers.

11)what is the downside of having activations with a standard deviation too far away from 1 ?

Ans : normal distribution with a mean of 0 and a standard deviation of 1 is called a standard normal distribution. Areas of the normal distribution are often represented by tables of the standard normal distribution. A portion of a table of the standard normal distribution .

my standard deviation and variance are above 1, the standard deviation will be smaller than the variance. But if they are below 1, the standard deviation will be bigger than the variance.

in a normal distribution, a score that is 1 s.d. above the mean is equivalent to the 84th percentile

12) how can weight initialization help to Avoid this problems ?

Ans : The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network.