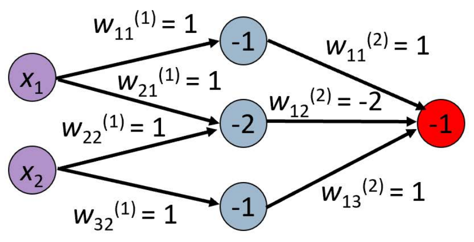
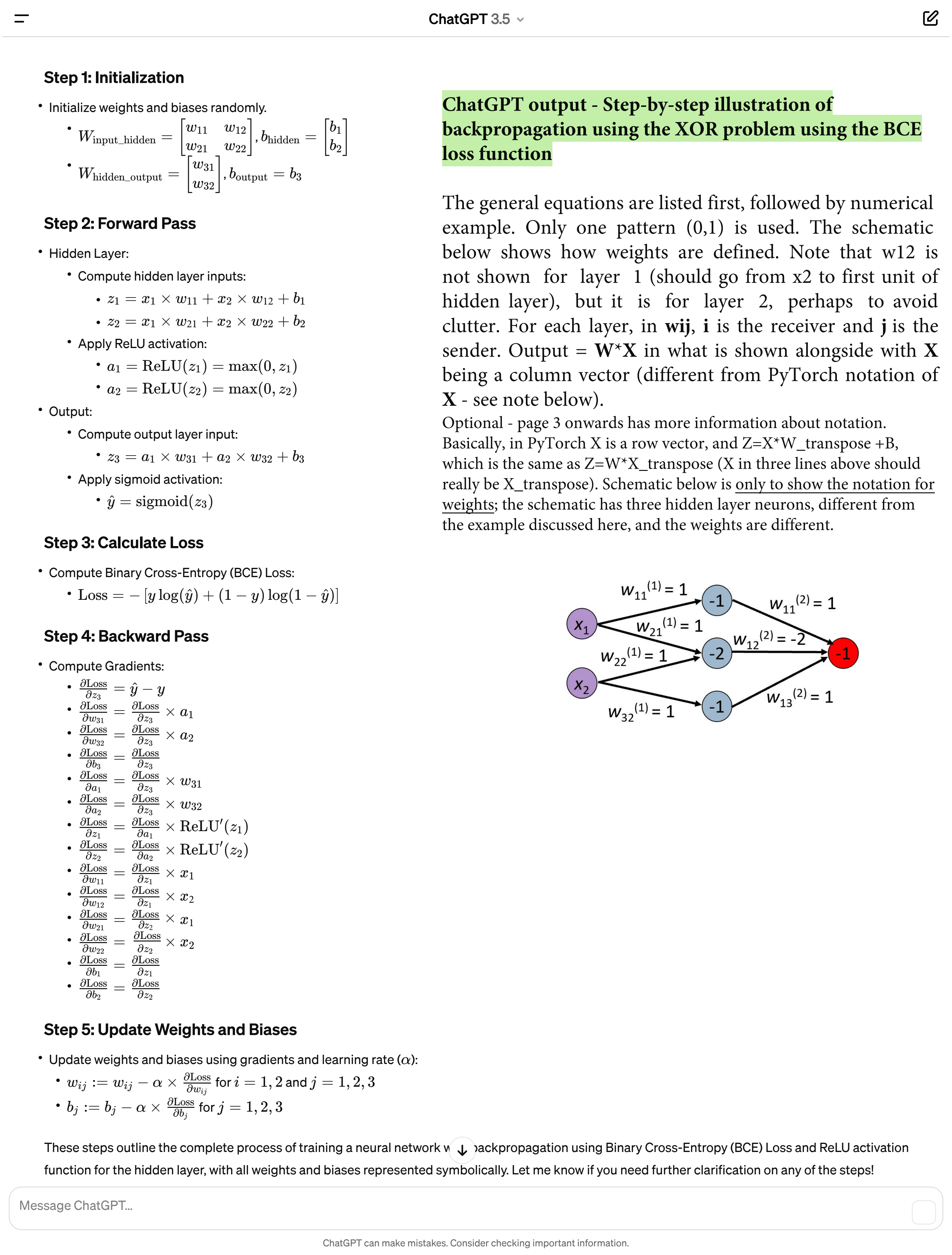
ChatGPT output - Step-by-step illustration of backpropagation using the XOR problem using the BCE loss function

The general equations are listed first, followed by numerical example. Only one pattern (0,1) is used. The schematic below shows how weights are defined. Note that w12 is not shown for layer 1 (should go from x2 to first unit of hidden layer), but it is for layer 2, perhaps to avoid clutter. For each layer, in **wij**, **i** is the receiver and **j** is the sender. Output = **W*X** in what is shown alongside with **X** being a column vector (different from PyTorch notation of **X** - see note below).
Optional - page 3 onwards has more information about notation. Basically, in PyTorch X is a row vector, and Z=X*W_transpose +B, which is the same as Z=W*X_transpose (X in three lines above should really be X_transpose). Schematic below is only to show the notation for weights; the schematic has three hidden layer neurons, different from the example discussed here, and the weights are different.




ChatGPT 3.5

Step 1: Initialization

- Initialize weights and biases randomly.
  - $W_{\text{input\_hidden}} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}, b_{\text{hidden}} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$
  - $W_{\text{hidden\_output}} = \begin{bmatrix} w_{31} \\ w_{32} \end{bmatrix}, b_{\text{output}} = b_3$

Step 2: Forward Pass

- Hidden Layer:
  - Compute hidden layer inputs:
    - $z_1 = x_1 \times w_{11} + x_2 \times w_{12} + b_1$
    - $z_2 = x_1 \times w_{21} + x_2 \times w_{22} + b_2$
  - Apply ReLU activation:
    - $a_1 = \text{ReLU}(z_1) = \max(0, z_1)$
    - $a_2 = \text{ReLU}(z_2) = \max(0, z_2)$
- Output:
  - Compute output layer input:
    - $z_3 = a_1 \times w_{31} + a_2 \times w_{32} + b_3$
  - Apply sigmoid activation:
    - $\hat{y} = \text{sigmoid}(z_3)$

Step 3: Calculate Loss

- Compute Binary Cross-Entropy (BCE) Loss:
  - $\text{Loss} = -\left[y \log(\hat{y}) + (1 - y)\log(1 - \hat{y})\right]$

Step 4: Backward Pass

- Compute Gradients:
  - $\frac{\partial \text{Loss}}{\partial z_3} = \hat{y} - y$
  - $\frac{\partial \text{Loss}}{\partial w_{31}} = \frac{\partial \text{Loss}}{\partial z_3} \times a_1$
  - $\frac{\partial \text{Loss}}{\partial w_{32}} = \frac{\partial \text{Loss}}{\partial z_3} \times a_2$
  - $\frac{\partial \text{Loss}}{\partial b_3} = \frac{\partial \text{Loss}}{\partial z_3}$
  - $\frac{\partial \text{Loss}}{\partial a_1} = \frac{\partial \text{Loss}}{\partial z_3} \times w_{31}$
  - $\frac{\partial \text{Loss}}{\partial a_2} = \frac{\partial \text{Loss}}{\partial z_3} \times w_{32}$
  - $\frac{\partial \text{Loss}}{\partial z_1} = \frac{\partial \text{Loss}}{\partial a_1} \times \text{ReLU}'(z_1)$
  - $\frac{\partial \text{Loss}}{\partial z_2} = \frac{\partial \text{Loss}}{\partial a_2} \times \text{ReLU}'(z_2)$
  - $\frac{\partial \text{Loss}}{\partial w_{11}} = \frac{\partial \text{Loss}}{\partial z_1} \times x_1$
  - $\frac{\partial \text{Loss}}{\partial w_{12}} = \frac{\partial \text{Loss}}{\partial z_1} \times x_2$
  - $\frac{\partial \text{Loss}}{\partial w_{21}} = \frac{\partial \text{Loss}}{\partial z_2} \times x_1$
  - $\frac{\partial \text{Loss}}{\partial w_{22}} = \frac{\partial \text{Loss}}{\partial z_2} \times x_2$
  - $\frac{\partial \text{Loss}}{\partial b_1} = \frac{\partial \text{Loss}}{\partial z_1}$
  - $\frac{\partial \text{Loss}}{\partial b_2} = \frac{\partial \text{Loss}}{\partial z_2}$

Step 5: Update Weights and Biases

- Update weights and biases using gradients and learning rate (α):
  - $w_{ij} := w_{ij} - \alpha \times \frac{\partial \text{Loss}}{\partial w_{ij}}$ for $i = 1, 2$ and $j = 1, 2, 3$
  - $b_j := b_j - \alpha \times \frac{\partial \text{Loss}}{\partial b_j}$ for $j = 1, 2, 3$

These steps outline the complete process of training a neural network w backpropagation using Binary Cross-Entropy (BCE) Loss and ReLU activation function for the hidden layer, with all weights and biases represented symbolically. Let me know if you need further clarification on any of the steps!

Message ChatGPT...

ChatGPT can make mistakes. Consider checking important information.

Step 1: Initialization

- Initialize weights and biases randomly.
  - $W_{\text{input\_hidden}} = \begin{bmatrix} 0.4 & -0.2 \\ 0.1 & 0.6 \end{bmatrix}, b_{\text{hidden}} = \begin{bmatrix} 0.3 \\ -0.4 \end{bmatrix}$
  - $W_{\text{hidden\_output}} = \begin{bmatrix} -0.5 \\ 0.8 \end{bmatrix}, b_{\text{output}} = -0.7$

Step 2: Forward Pass

- Hidden Layer:
  - Compute hidden layer inputs:
    - $z_1 = 0 \times 0.4 + 1 \times (-0.2) + 0.3 = 0.1$
    - $z_2 = 0 \times 0.1 + 1 \times 0.6 + (-0.4) = 0.2$
  - Apply ReLU activation:
    - $a_1 = \text{ReLU}(z_1) = \max(0, 0.1) = 0.1$
    - $a_2 = \text{ReLU}(z_2) = \max(0, 0.2) = 0.2$
- Output:
  - Compute output layer input:
    - $z_3 = a_1 \times (-0.5) + a_2 \times 0.8 + (-0.7) = -0.59$
  - Apply sigmoid activation:
    - $\hat{y} = \text{sigmoid}(z_3) \approx \text{sigmoid}(-0.59) \approx 0.3566$

$$\begin{aligned}\sigma'(x) &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1+e^{-x}-1}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} \\ &= \frac{1}{(1+e^{-x})} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\ &= \sigma(x)\,(1-\sigma(x))\end{aligned}$$

$$L(\boldsymbol{W}) = -[\boldsymbol{y} * \log(\hat{\boldsymbol{y}}) + (1-\boldsymbol{y}) * \log(1-\hat{\boldsymbol{y}})]$$

$$\frac{\partial L(\boldsymbol{W})}{\partial \hat{\boldsymbol{y}}} = -\left(\frac{\boldsymbol{y}}{\hat{\boldsymbol{y}}} - \frac{1-\boldsymbol{y}}{1-\hat{\boldsymbol{y}}}\right) = \frac{\hat{\boldsymbol{y}} - \boldsymbol{y}}{\hat{\boldsymbol{y}}(1-\hat{\boldsymbol{y}})}$$

Step 3: Calculate Loss

- Compute binary cross-entropy loss:
  - $\text{Loss} = -[1 \log(\hat{y}) + (1-1)\log(1-\hat{y})]$= -(-1.03) = 1.0310

Step 4: Backward Pass

- Compute Gradients:
  - $\frac{\partial \text{Loss}}{\partial z_3} = \hat{y} - y = 0.3566 - 1 = -0.64$ Note: [del_L/del_y_hat]*[del_y_hat/del_z3] = y_hat-y
  - $\frac{\partial \text{Loss}}{\partial w_{31}} = \frac{\partial \text{Loss}}{\partial z_3} \times a_1 = -0.643 \times 0.1 = -0.06434$
  - $\frac{\partial \text{Loss}}{\partial w_{32}} = \frac{\partial \text{Loss}}{\partial z_3} \times a_2 = -0.643 \times 0.2 = -0.1287$
  - $\frac{\partial \text{Loss}}{\partial b_3} = \frac{\partial \text{Loss}}{\partial z_3} = -0.6434$
  - $\frac{\partial \text{Loss}}{\partial a_1} = \frac{\partial \text{Loss}}{\partial z_3} \times w_{31} = -0.6434 \times (-0.5) = 0.3217$
  - $\frac{\partial \text{Loss}}{\partial a_2} = \frac{\partial \text{Loss}}{\partial z_3} \times w_{32} = -0.6434 \times 0.8 = -0.5147$
  - $\frac{\partial \text{Loss}}{\partial z_1} = \frac{\partial \text{Loss}}{\partial a_1} \times \text{ReLU}'(z_1) = 0.3217 \times 1 = 0.3217$
  - $\frac{\partial \text{Loss}}{\partial z_2} = \frac{\partial \text{Loss}}{\partial a_2} \times \text{ReLU}'(z_2) = -0.5147 \times 1 = -0.5147$
  - $\frac{\partial \text{Loss}}{\partial w_{11}} = \frac{\partial \text{Loss}}{\partial z_1} \times x_1 = 0.3217 \times 0 = 0$
  - $\frac{\partial \text{Loss}}{\partial w_{12}} = \frac{\partial \text{Loss}}{\partial z_1} \times x_2 = 0.3217 \times 1 = 0.3217$
  - $\frac{\partial \text{Loss}}{\partial w_{21}} = \frac{\partial \text{Loss}}{\partial z_2} \times x_1 = -0.5147 \times 0 = 0$
  - $\frac{\partial \text{Loss}}{\partial w_{22}} = \frac{\partial \text{Loss}}{\partial z_2} \times x_2 = -0.5147 \times 1 = -0.5147$
  - $\frac{\partial \text{Loss}}{\partial b_1} = \frac{\partial \text{Loss}}{\partial z_1} = 0.3217$
  - $\frac{\partial \text{Loss}}{\partial b_2} = \frac{\partial \text{Loss}}{\partial z_2} = -0.5147$

Step 5: Update Weights and Biases

- Update weights and biases using gradients and learning rate ($\alpha = 0.1$):
  - $w_{31} := -0.5 - 0.1 \times (-0.06434) = -0.4936$
  - $w_{32} := 0.8 - 0.1 \times (-0.1287) = 0.8129$
  - $b_3 := -0.7 - 0.1 \times (-0.6434) = -0.6357$
  - $w_{11} := 0.4 - 0.1 \times 0 = 0.4$
  - $w_{12} := 0.1 - 0.1 \times 0 = 0.1$
  - $w_{21} := -0.2 - 0.1 \times 0.3217 = -0.23217$
  - $w_{22} := 0.6 - 0.1 \times (-0.5147) = 0.6515$
  -

Pytorch and neural network conventions and notations

My personal notes used to compare notations and conventions between Pytorch tensors and regular matrix math

DANIEL COELHO
DEC 17, 2022

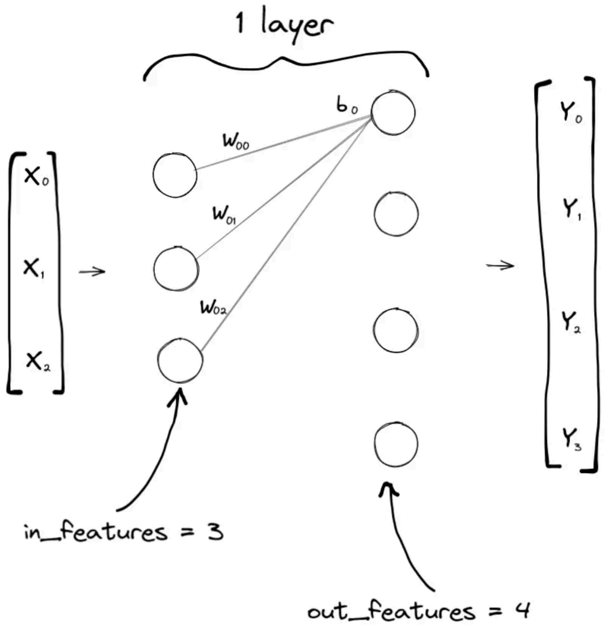
Prologue

I've been getting into machine learning with Pytorch these past few months, and one of my notes which has gotten the most mileage is this "deconfuser" note where I write out all the useful conventions that I need, but occasionally forget. I figured there's a chance somebody finds them useful, and it would be a good simple post to get some traction and get back to blogging more.

Here it is:

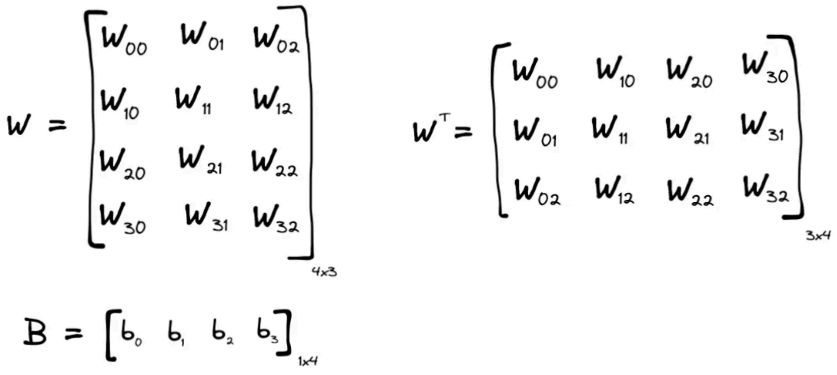
Conventions

Imagine a neural network with a single 3x4 linear layer:



Note that it has two columns of circles ("neurons"), but it is a *single* layer: Its easier to think of the layer as the thing between the neurons instead of an actual layer of neurons (which would be one of those columns).

You can describe the layer's weights and biases with matrices `W` and `B` like this:



$$W = \begin{bmatrix} W_{00} & W_{01} & W_{02} \\ W_{10} & W_{11} & W_{12} \\ W_{20} & W_{21} & W_{22} \\ W_{30} & W_{31} & W_{32} \end{bmatrix}_{4\times3} \qquad W^{T} = \begin{bmatrix} W_{00} & W_{10} & W_{20} & W_{30} \\ W_{01} & W_{11} & W_{21} & W_{31} \\ W_{02} & W_{12} & W_{22} & W_{32} \end{bmatrix}_{3\times4}$$

$$B = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix}_{1\times4}$$

Notice their shapes on the bottom right (number of rows x number of columns). Sometimes you will see `W` drawn transposed instead, like how I've also drawn it on the right in the above image (in this case with 3 rows and 4 columns).

This is how you describe that layer in Pytorch:

```
import torch

layer = torch.nn.Linear(in_features=3, out_features=4, bias=True)
print(layer.weight.shape)  # Prints [4, 3]
print(layer.bias.shape)    # Prints [4]
```

Note that it has a 4x3 matrix of weights, which is why I chose to draw `W` in that way. Also note that technically the bias is a line vector (it has only one dimension with 4 values) instead of being a 1x4 row vector. It's more useful to think of these as 1xN row vectors instead of line vectors though, for reasons we'll get to at the Broadcasting section below.

The layer receives a 3-dimensional input X, and produces a 4-dimensional output Y. Again, they're drawn vertically (kinda like column vectors) on the neural network diagram because it sort of fits, but in Pytorch we'd rather think of them as row vectors.

$$X = \begin{bmatrix} X_0 & X_1 & X_2 \end{bmatrix}_{1x3}$$

$$Y = \begin{bmatrix} Y_0 & Y_1 & Y_2 & Y_3 \end{bmatrix}_{1x4}$$

This is how you apply that layer to a tensor X and receive a tensor Y in Pytorch:

```
X = torch.rand(3)
layer = torch.nn.Linear(in_features=3, out_features=4, bias=True)

# The generic way of applying any layer
Y = layer(X)

# What torch.nn.Linear does internally
Y_manual = X @ layer.weight.transpose(0, 1) + layer.bias

assert (Y == Y_manual).all().item()  # y and y_manual are the exact same
```

The @ operator actually just maps to torch.matmul, which is a regular matrix multiply.

The `.transpose(0, 1)` just transposes the dimensions 0 and 1, so rows with columns. Drawing it out, Y and `Y_manual` are both computed by the simple linear layer function:

$$Y = x * W^T + B$$

$$\begin{bmatrix} Y_0 & Y_1 & Y_2 & Y_3 \end{bmatrix}_{1x4} = \begin{bmatrix} X_0 & X_1 & X_2 \end{bmatrix}_{1x3} * \begin{bmatrix} W_{00} & W_{10} & W_{20} & W_{30} \\ W_{01} & W_{11} & W_{21} & W_{31} \\ W_{02} & W_{12} & W_{22} & W_{32} \end{bmatrix}_{3x4} + \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix}_{1x4}$$

According to the matrix multiply rules, you'll see you can perform the multiplication of matrices with shapes [1, 3] * [3, 4], (as the two numbers closer together are the same (3)) and that it will lead to a result with shape [1, 4] (the two numbers that are further apart), so that everything works out. Well, except for how X is not really a 1x3 row vector and is *really* a line vector... I think it's time:

Broadcasting

I've been saying that X, Y and B are row vectors so far, but in Pytorch they're 1-dimensional line vectors. Who's to say they don't represent a column instead of a row?

Well Pytorch has this mechanism called "broadcasting", where tensors can receive extra dimensions to make their shapes match up with other tensor shapes, in order to be able to perform some operation on them.

For example, look at this:

```
t1 = torch.tensor([1, 2, 3])                     # Shape [3] <--- line vector
t2 = torch.tensor([[10, 20, 30], [40, 50, 60]])  # Shape [2, 3]  <--- 2D matrix

s = t1 + t2
print(s)         # Prints [[11, 22, 33], [41, 52, 63]]
print(s.shape)   # Prints [2, 3]  <--- 2D matrix
```

Note that it "broadcast" (copy-pasted) the `t1` tensor values into two identical rows of a "temporary tensor" `[[1, 2, 3], [1, 2, 3]]` that could then be added with `t2` element-wise.

To know if the broadcast mechanism can help your case, you can follow these rules of thumb:

1. Align tensor shapes to the right:

```
t2: [2, 3]
t1:    [3]
```

2. Expand the tensor with fewer dimensions to have "1" for all the other dimensions:

```
t2: [2, 3]
t1: [1, 3]
```

3. Pytorch will "broadcast" if and only if for each dimension you have an equal number of values (like how we have "3" for the right-most dimension there), or one of the tensors has just 1 value (like how we have 2 and 1 for the left-most dimension there). If one of the tensors has a single value for a dimension, Pytorch will just copy-paste that value until that the two tensors end up with the same number of values for that particular dimension (which is what we saw on the snippet above when we did `s = t1 + t2`)

This is why I said 1-dimensional vectors are kind of the same as row-vectors: Pytorch will broadcast them from being [N] to [1, N] before you try performing an operation (like a matrix multiply).

This is why we could do this:

```
layer = torch.nn.Linear(in_features=3, out_features=4, bias=True)

X = torch.rand(3)                  # Shape    [3]
Wt = layer.weight.transpose(0, 1)  # Shape [3, 4]
B = layer.bias                     # Shape    [4]

Y_manual =        X @ Wt     + B
# Shapes:       [3] @ [3, 4] + [4]
# Shapes:    [1, 3] @ [3, 4] + [4]     (after broadcasting X)
# Shapes:         [1, 4]     + [4]     (after matrix multiply)
# Shapes:         [1, 4]     + [1, 4]  (after broadcasting B)
# Shapes:                [1, 4]        (after adding B)
```

Batch matrix multiplication

There's one last thing I think still fits in this post: In Pytorch models you rarely have a single 2D matrix for a tensor: You'll have many more dimensions (batch, channels, etc.). What actually happens if we do something like this?

```
X = torch.rand((10, 2, 3))  # Shape [10, 2, 3]

# Like before, this has a [3, 4] weight tensor and a [4] bias tensor
layer = torch.nn.Linear(in_features=3, out_features=4, bias=True)

# These are all identical, and end up with shape [10, 2, 4]
Y = layer(X)
Y_manual = X @ layer.weight.transpose(0, 1) + layer.bias
Y_matmul = torch.matmul(X, layer.weight.transpose(0, 1)) + layer.bias
```

You can think of that 3-dimensional `X` as being 10 groups of [2, 3] matrices. When you apply `layer` (or use the @ operator, or call `torch.matmul`), Pytorch is going to broadcast `layer`'s weight and biases, and then perform each of the ten [2, 3] * [3, 4] matrix multiplies independently, returning a `[10, 2, 4]` tensor.

There is also a dedicated torch.bmm function for performing the **b**atch **m**atrix **m**ultiply. However, annoyingly, this function doesn't perform broadcasting and only works when the two arguments are tensors with exactly 3 dimensions:

```
layer = torch.nn.Linear(in_features=3, out_features=4, bias=True)

X   = torch.rand((20, 2, 3))        # Shape [20, 2, 3]
Wt  = layer.weight.transpose(0, 1)  # Shape     [3, 4]
Wte = Wt.expand(20, 3, 4)           # Shape [20, 3, 4]

Y = torch.bmm(X, Wt)                # Raises an error
Y = torch.bmm(X, Wte)               # OK
```

Finally, what happens if you do a matmul with both tensor arguments having more than 2 dimensions?

```
A = torch.tensor([  # Shape [2, 2, 3]
    [[1, 2, 3],
     [4, 5, 6]],

    [[7, 8, 9],
     [10, 11, 12]]
])

B = torch.tensor([  # Shape [2, 3, 4]
    [[2, 2, 2, 2],
     [3, 3, 3, 3],
     [4, 4, 4, 4]],

    [[2, 2, 2, 2],
     [3, 3, 3, 3],
     [4, 4, 4, 4]],
])

C = A @ B           # Shape [2, 2, 4]
# C corresponds to torch.tensor([
#    [[ 20,  20,  20,  20]
#     [ 47,  47,  47,  47]],
#
#    [[ 74,  74,  74,  74],
#     [101, 101, 101, 101]]
#])
assert (C[0] == A[0] @ B[0]).all().item()
assert (C[1] == A[1] @ B[1]).all().item()
```

Like the `asserts` suggest, it just does 2-dimensional matrix multiplies pairwise for the two arguments, regardless of how many higher dimensions they have. This means you must have exactly the same number of values in each of the higher dimensions though, and something like this wouldn't work:

```
A = torch.tensor([  # Shape [2, 2, 3]
    [[1, 2, 3],
     [4, 5, 6]],

    [[7, 8, 9],
     [10, 11, 12]]
])

B_diff = torch.tensor([  # Shape [3, 3, 4]
    [[2, 2, 2, 2],
     [3, 3, 3, 3],
     [4, 4, 4, 4]],

    [[2, 2, 2, 2],
     [3, 3, 3, 3],
     [4, 4, 4, 4]],

    [[2, 2, 2, 2],
     [3, 3, 3, 3],
     [4, 4, 4, 4]],
])
D = A @ B_diff  # Raises an error, since A and B_diff have incompatible shapes
```

Broadcasting would also help you in this case though, so you could perform this batch matrix multiplication in these cases:

```
A = torch.rand((2, 2, 3))  # Theses are the shapes of the tensors
B = torch.rand((2, 3, 4))
C = A @ B  # Ok

A = torch.rand((2, 2, 3))
B = torch.rand(   (3, 4))
C = A @ B  # Ok

A = torch.rand((2, 2, 3))
B = torch.rand((1, 3, 4))
C = A @ B  # Ok

A = torch.rand((1, 2, 2, 3))
B = torch.rand(   (1, 3, 4))
C = A @ B  # Ok

A = torch.rand((1, 2, 2, 3))
B = torch.rand((4, 1, 3, 4))
C = A @ B  # Ok

A = torch.rand((2, 2, 2, 3))
B = torch.rand((4, 1, 3, 4))
C = A @ B  # Error

A = torch.rand((1, 2, 2, 3))
```

```
B = torch.rand(   (4, 3, 4))
C = A @ B  # Error

A = torch.rand((1, 2, 2, 3))
B = torch.rand(      (3, 4))
C = A @ B  # Ok
```

Epilogue

That's about as much as I wanted to cover on this one. Some of these things always confused me, like how you specify a linear layer like `torch.nn.Linear(3, 4)` and it actually has a 4x3 weights matrix, but transposes it for the multiply.

I deliberately chose different numbers of values in all dimensions, but if you have square matrices (more often than not) you can see how this can lead to subtle issues and confusion.

Thanks for reading!

Daniel's blog is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

Type your email... Subscribe