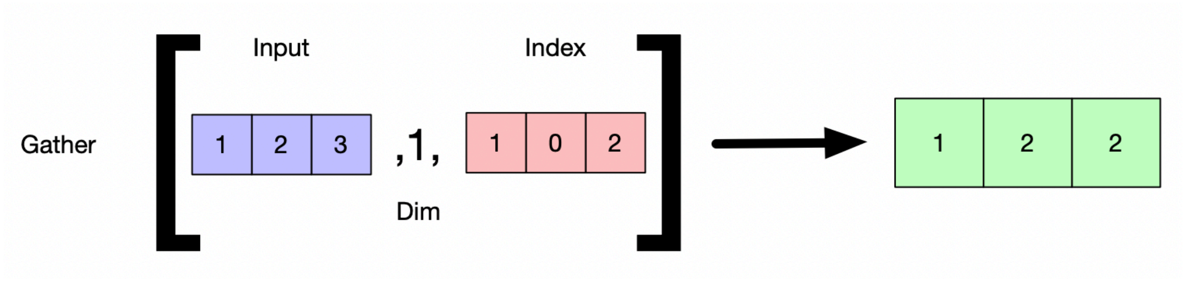
Gather/Scatter On Pytorch

Those operations are used for selecting/distributing values on a tensor, but in a fast way. For example implementing Cross-Entropy using gather has good performance, unfortunately the documentation of those functions are not very explanatory.

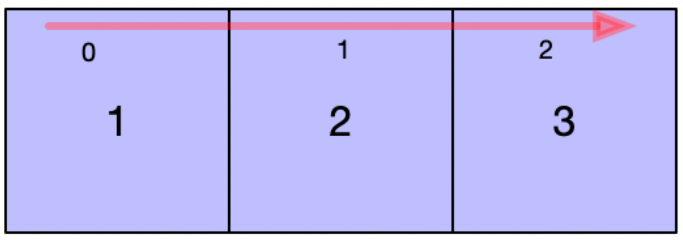
Gather

Get values from input tensor along a dimension dim using some indexes. Both input and index needs to have the same shape.

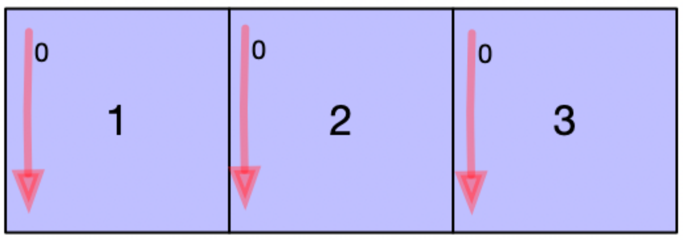
Gather in 1D



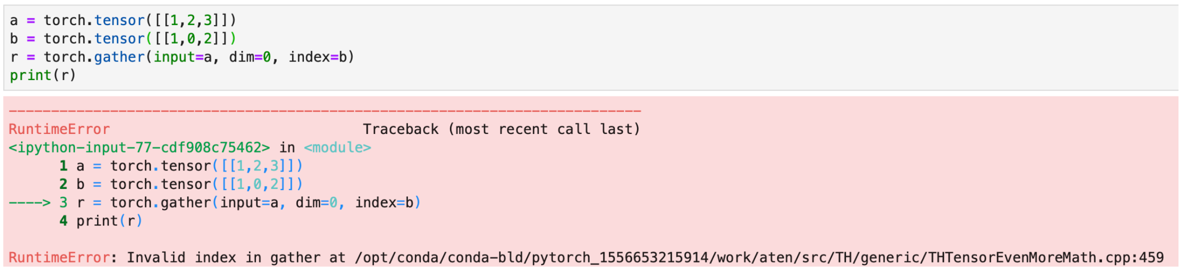
Gathering along the dimension 1, means that we're addressing the input like this:



Doing the same on the dimension zero would have a completely different effect, on this case we would not even be able to address the input properly because there is no index besides 0



An important point to observe is that the index reset to 0 at the end of the column. And yes it's confusing but dimension 0 scan columwise while dimension 1 scan rowise.

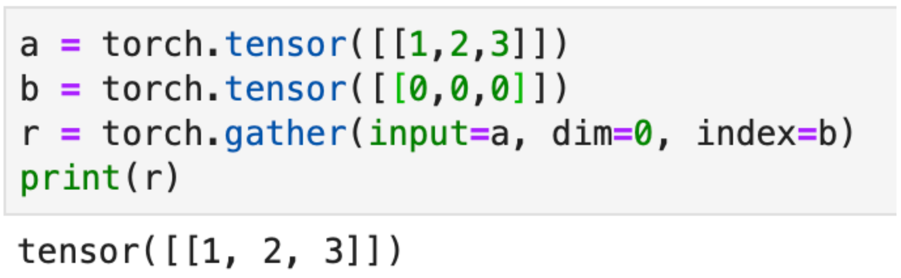


```
a = torch.tensor([[1,2,3]])
b = torch.tensor([[1,0,2]])
r = torch.gather(input=a, dim=0, index=b)
print(r)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-77-cdf908c75462> in <module>
      1 a = torch.tensor([[1,2,3]])
      2 b = torch.tensor([[1,0,2]])
----> 3 r = torch.gather(input=a, dim=0, index=b)
      4 print(r)

RuntimeError: Invalid index in gather at /opt/conda/conda-bld/pytorch_1556653215914/work/aten/src/TH/generic/THTensorEvenMoreMath.cpp:459
```

The only way this could work would be to use zeros everywhere on the index.

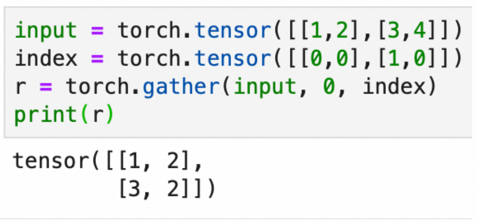


```
a = torch.tensor([[1,2,3]])
b = torch.tensor([[0,0,0]])
r = torch.gather(input=a, dim=0, index=b)
print(r)
```

```
tensor([[1, 2, 3]])
```

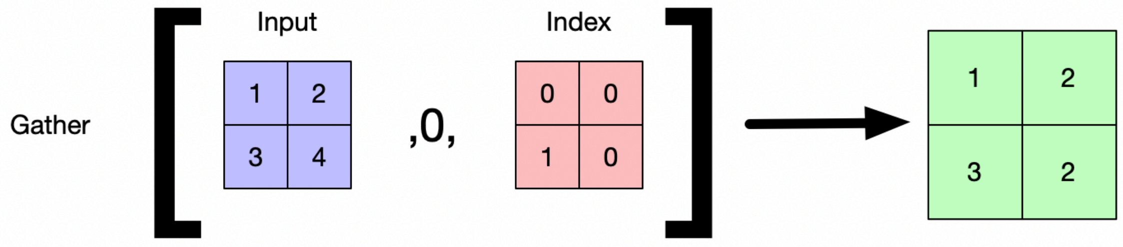
Gather in 2D

In 2D things get more interesting, here you can consider the index as a placeholder with same shape as input where every cell on the index it's a pointer to where to gather on the input, consider the following example:

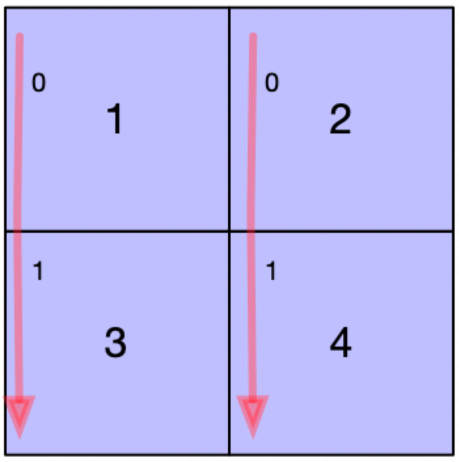


```
input = torch.tensor([[1,2],[3,4]])
index = torch.tensor([[0,0],[1,0]])
r = torch.gather(input, 0, index)
print(r)
```

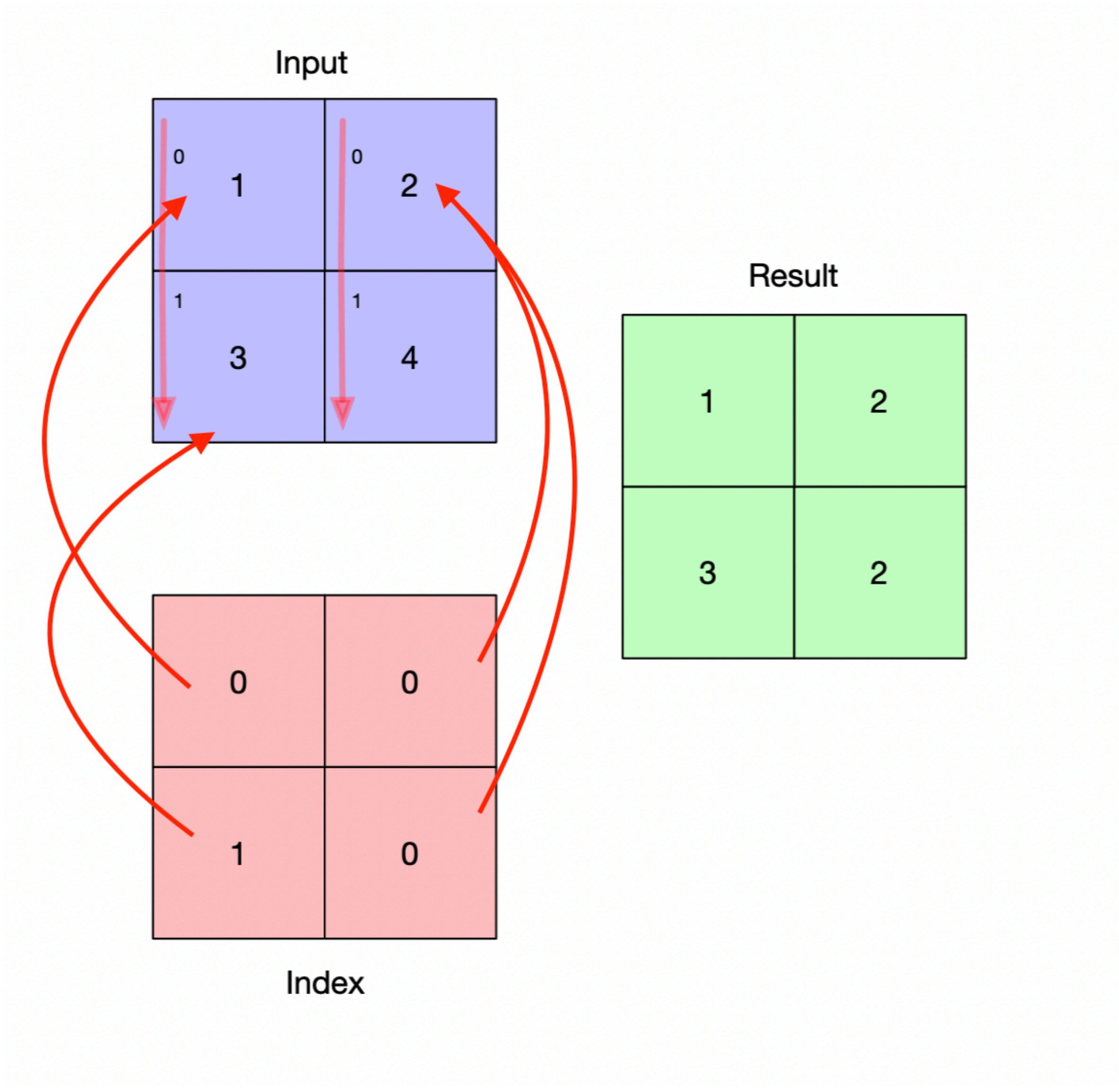
```
tensor([[1, 2],
        [3, 2]])
```



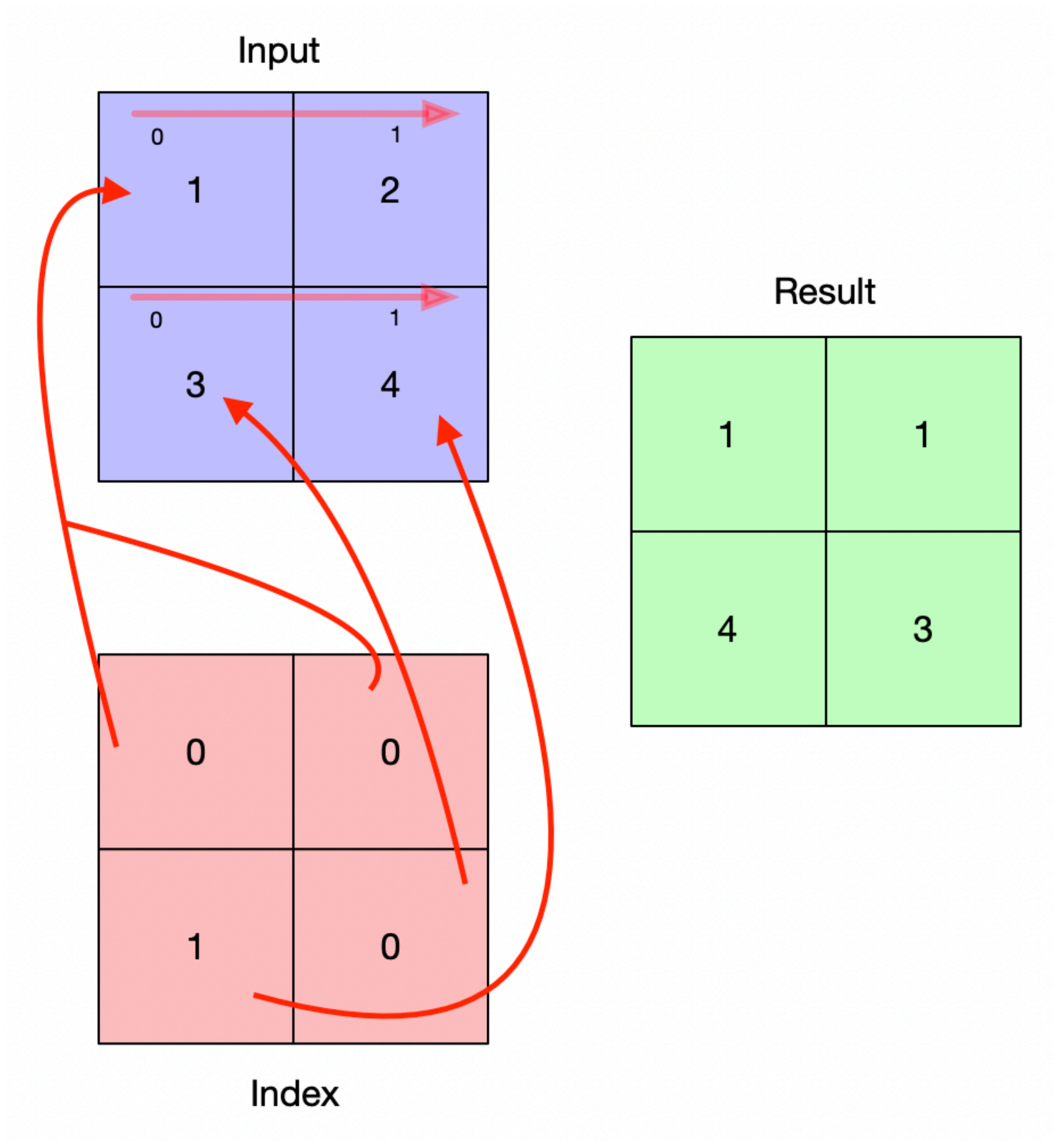
Here gathering along the dimension zero means that we're addressing the input like this:



So if we put together the index matrix we would have this:



Now consider that we're choose the dimension 1



In practice people reshape the inputs to a vector to make things more manageable in their heads.

Scatter

This operation is used to scatter values along a tensor following the same rules as gather, but now pushing values instead of selecting them.

```
x = torch.arange(3).view(1,3)+10
print('input:',x)
y = torch.zeros_like(x)
# index need to have same shape as x
b = torch.tensor([[0,2,1]])
print('index:',b)
# Scatter on y, the values on x along dimension 1
y.scatter(dim=1, index=b, src=x)
```

```
input: tensor([[10, 11, 12]])
index: tensor([[0, 2, 1]])

tensor([[10, 12, 11]])
```

Why this?

- Because it's faster than doing in a for-loop
- Pytorch uses same semantic for distributing tensors across different machines/GPUs

Example of CrossEntropy

Here we implement a CrossEntropy with gather, one cool feature is that gather accept broadcast so the input could be [batch x classes] while the index [batch x 1]

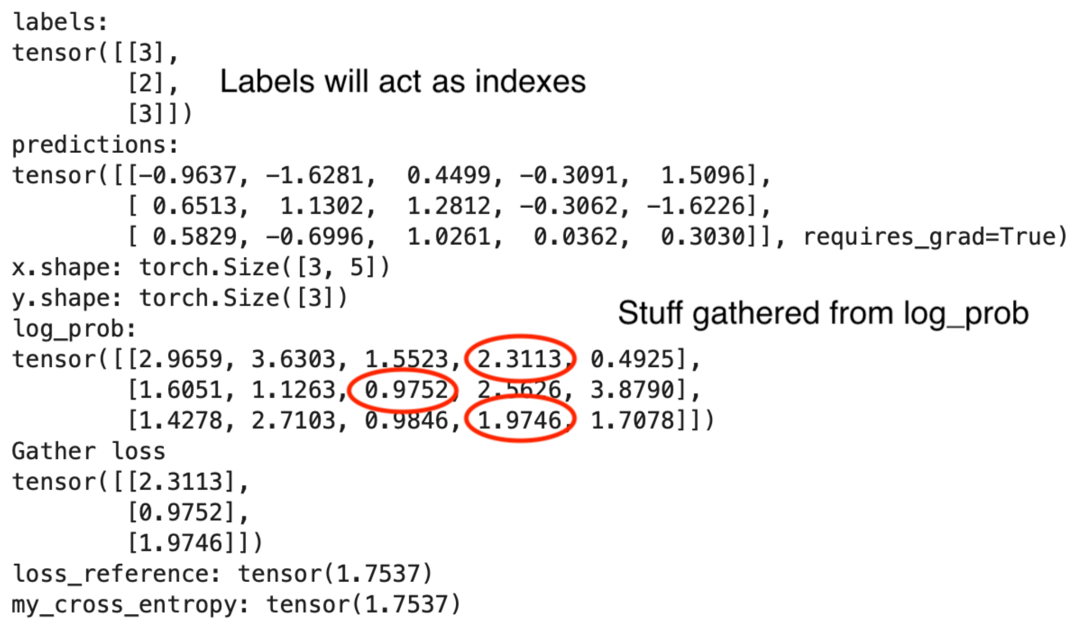
```
# Define custom cross_entropy
# x: shape [batch x C]
# y: shape [batch]
def my_cross_entropy(x, target):
    # Calculate the log-probability of x along first dimension
    log_prob = -1.0 * F.log_softmax(x, 1)
    print('log_prob:')
    print(log_prob)
    # Unsqueze will make y shape become [batch x 1]
    # Gather elements from log_probability along second dimension (dim=1)
    # here log_prob.shape [ batch x C]
    # target.shape [ batch x 1], so the shapes wont match but pytorch will broadcast
    loss_no_gather = log_prob[range(log_prob.shape[0]), target]
    print('No gather loss(SLOW):')
    print(loss_no_gather.unsqueeze(1))
    loss = log_prob.gather(1, target.unsqueeze(1))
    print('Gather loss')
    print(loss)
    loss = loss.mean()
    return loss

# Reference CrossEntropy
criterion = nn.CrossEntropyLoss()
```

Inspecting closer what's going on you can observe:

- Labels after unsqueeze(1) will become [batch x 1] which doesn't exactly match with [batch x C]

- Gather will broadcast and select the correct index for each element on the batch along the dimension 1
- Both custom Cross_Entropy and Reference Cross entropy return the same value
- The other option would be loss_no_gather = log_prob[range(log_prob.shape[0]), target] (SLOW because doesn't use pytorch ops)



```
labels:
tensor([[3],
        [2],
        [3]])
predictions:
tensor([[-0.9637, -1.6281,  0.4499, -0.3091,  1.5096],
        [ 0.6513,  1.1302,  1.2812, -0.3062, -1.6226],
        [ 0.5829, -0.6996,  1.0261,  0.0362,  0.3030]], requires_grad=True)
x.shape: torch.Size([3, 5])
y.shape: torch.Size([3])
log_prob:
tensor([[2.9659, 3.6303, 1.5523, 2.3113, 0.4925],
        [1.6051, 1.1263, 0.9752, 2.5626, 3.8790],
        [1.4278, 2.7103, 0.9846, 1.9746, 1.7078]])
Gather loss
tensor([[2.3113],
        [0.9752],
        [1.9746]])
loss_reference: tensor(1.7537)
my_cross_entropy: tensor(1.7537)
```

References

- https://discuss.pytorch.org/t/convert-int-into-one-hot-format/507/3
- https://discuss.pytorch.org/t/custom-loss-functions/29387/8
- https://stackoverflow.com/questions/50999977/what-does-the-gather-function-do-in-pytorch-in-layman-terms