Pay attention that, when it comes to (2D) CNN, we normalize batch_size * height * width over each channel. So that gamma and beta have the lengths the same as channel_count. In our implementation, we need to manually reshape gamma and beta so that they could (be automatically broadcast and) multipy the matrices in the desired way.

```
N, C, H, W = X.shape
# mini-batch mean
mean = nd.mean(X, axis=(0, 2, 3))
```


11.2.6 Batch Normalization

First introduced by Ioffe and Szgedy in their 2015 paper, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* [123], batch normalization layers (or `BN` for short), as the name suggests, are used to normalize the activations of a given input volume before passing it into the next layer in the network.

If we consider x to be our mini-batch of activations, then we can compute the normalized $\hat{x}$ via the following equation:

$$\hat{x_i} = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \tag{11.9}$$

During *training*, we compute the μ_β and σ_β over each mini-batch β, where:

$$\mu_\beta = \frac{1}{M}\sum_{i=1}^{m} x_i \qquad \sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \tag{11.10}$$

We set ε equal to a small positive value such as 1e-7 to avoid taking the square root of zero. Applying this equation implies that the activations leaving a batch normalization layer will have approximately zero mean and unit variance (i.e., zero-centered).

At *testing* time, we replace the mini-batch μ_β and σ_β with *running averages* of μ_β and σ_β computed during the training process. This ensures that we can pass images through our network and still obtain accurate predictions without being biased by the μ_β and σ_β from the final mini-batch passed through the network at training time.

Batch normalization has been shown to be *extremely effective* at reducing the number of epochs it takes to train a neural network. Batch normalization also has the added benefit of helping "stabilize" training, allowing for a larger variety of learning rates and regularization strengths. Using batch normalization doesn't alleviate the need to tune these parameters of course, but it *will* make your life easier by making learning rate and regularization less volatile and more straightforward to tune. You'll also tend to notice *lower final loss* and a *more stable loss curve* when using batch normalization in your networks.

The biggest drawback of batch normalization is that it can actually slow down the wall time it takes to train your network (even though you'll need fewer epochs to obtain reasonable accuracy) by 2-3x due to the computation of per-batch statistics and normalization.

That said, I recommend using batch normalization in *nearly every situation* as it does make a significant difference. As we'll see later in this book, applying batch normalization to our network architectures can help us prevent overfitting and allows us to obtain significantly higher classification accuracy in fewer epochs compared to the same network architecture *without* batch normalization.

So, Where Do the Batch Normalization Layers Go?

You've probably noticed in my discussion of batch normalization I've left out exactly *where* in the network architecture we place the batch normalization layer. According to the original paper by Ioffe and Szegedy [123], they placed their batch normalization (`BN`) *before* the activation:

> *"We add the BN transform immediately before the nonlinearity, by normalizing* $x = Wu + b$."

Using this scheme, a network architecture utilizing batch normalization would look like this:

```
INPUT => CONV => BN => RELU ...
```

However, this view of batch normalization doesn't make sense from a statistical point of view. In this context, a BN layer is normalizing the distribution of features coming out of a CONV layer. Some of these features may be negative, in which they will be clamped (i.e., set to zero) by a nonlinear activation function such as ReLU.

If we normalize *before* activation, we are essentially including the negative values inside the normalization. Our zero-centered features are then passed through the ReLU where we kill of any activations less than zero (which include features which may have not been negative *before* the normalization) – this layer ordering entirely defeats the purpose of applying batch normalization in the first place.

Instead, if we place the batch normalization *after* the ReLU we will normalize the positive valued features without statistically biasing them with features that would have otherwise not made it to the next CONV layer. In fact, François Chollet, the creator and maintainer of Keras confirms this point stating that the BN should come after the activation:

> *"I can guarantee that recent code written by Christian [Szegedy, from the BN paper] applies relu before BN. It is still occasionally a topic of debate, though."* [124]

It is unclear why Ioffe and Szegedy suggested placing the BN layer before the activation in their paper, but further experiments [125] as well as anecdotal evidence from other deep learning researchers [126] confirm that placing the batch normalization layer *after* the nonlinear activation yields higher accuracy and lower loss in nearly all situations.

Placing the BN after the activation in a network architecture would look like this:

```
INPUT => CONV => RELU => BN ...
```

I can confirm that in nearly all experiments I've performed with CNNs, placing the BN after the RELU yields slightly higher accuracy and lower loss. That said, take note of the word *"nearly"* – there have been a *very small* number of situations where placing the BN before the activation worked better, which implies that you should default to placing the BN after the activation, but may want to dedicate (at most) one experiment to placing the BN before the activation and noting the results.

After running a few of these experiments, you'll quickly realize that BN after the activation performs better and there are more important parameters to your network to tune to obtain higher classification accuracy. I discuss this in more detail in Section 11.3.2 later in this chapter.