Everything you wish to know about BatchNorm

Alvaro Durán Tovar Follow
Aug 31, 2019 · 3 min read

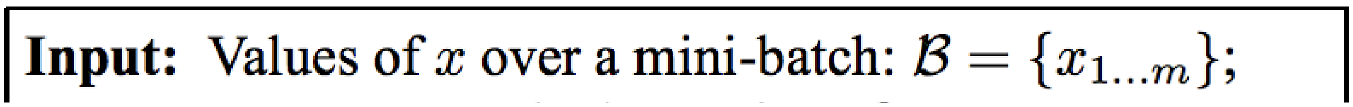
What is BatchNorm?

It's a type of neural network layer presented in 2015 by this paper. This layer have the following properties:

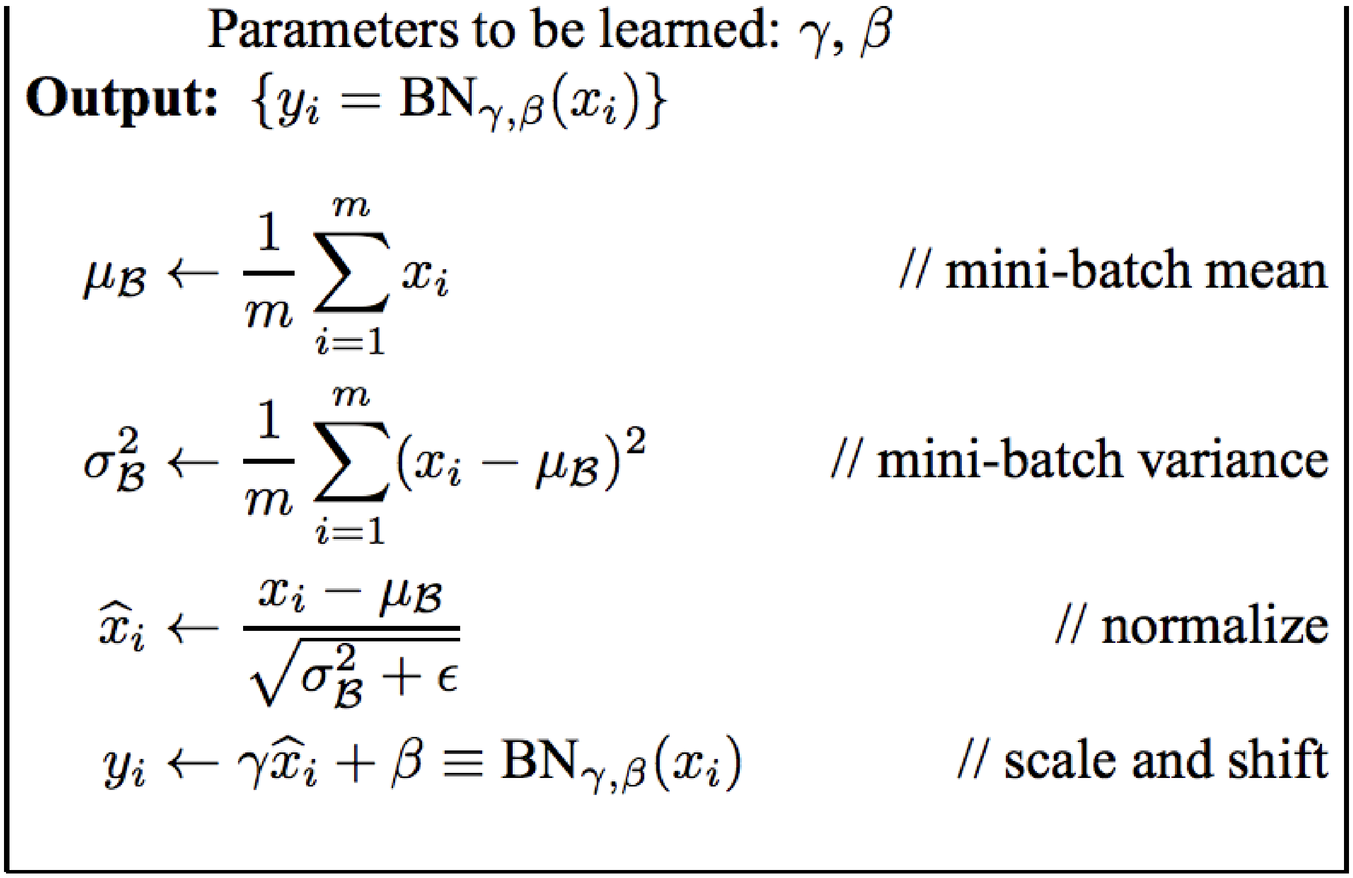
- **Faster training**: As the distribution of the weights of the network varies much less with this layer (this called *internal covariate shift* in the paper) we can use higher learning rates. The direction in which we are heading during training is less erratic allowing us to move faster on the direction of the loss.
- **Improves regularization**: Even though the network will see the same examples on each epoch, the normalization of each mini-batch is different, thus changing the values slightly each time. The meaning of the input is the same, but not how it's presented. The task is slightly more difficult for the network, rather than always seeing same input in the same way. That means we can reduce dropout thanks to this.
- **Improves accuracy**: Probably because of a combination of the previous two points the paper mentions that they got a better accuracy that state of the art results at that time.

How it works?

What BatchNorm does is to ensure that the received input have mean 0 and a standard deviation of 1. The algorithm as presented in the paper:



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;


Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

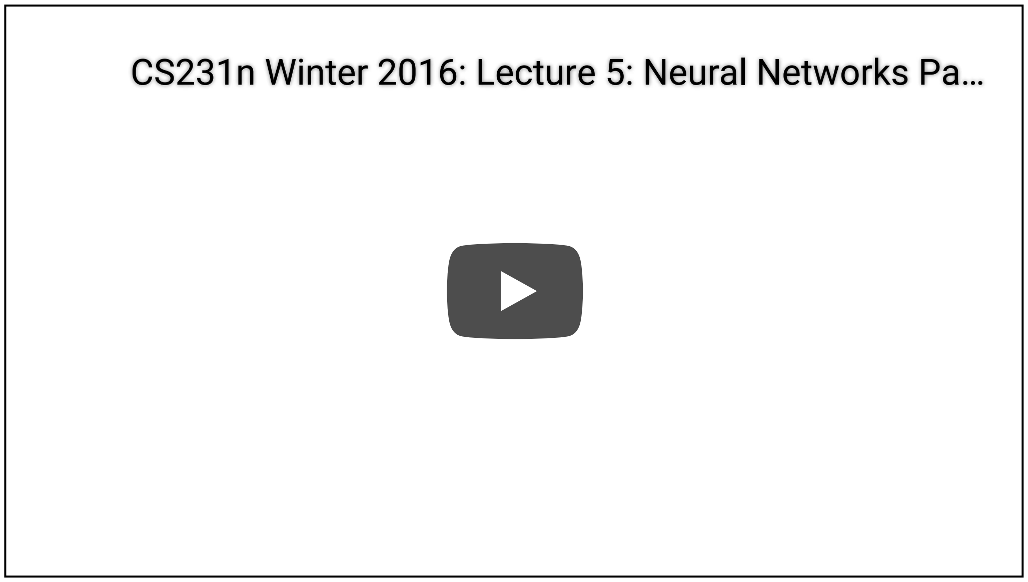
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Here is my own implementation of it in pytorch:

```
        self.register_buffer("moving_avg", torch.zeros(num_fea
tures))
        self.register_buffer("moving_var", torch.ones(num_feat
ures))
        self.register_buffer("eps", torch.tensor(eps))
        self.register_buffer("momentum", torch.tensor(momentum
))
        self._reset()

    def _reset(self):
        self.gamma.data.fill_(1)
        self.beta.data.fill_(0)

    def forward(self, x):
        if self.training:
            mean = x.mean(dim=0)
            var = x.var(dim=0)
            self.moving_avg = self.moving_avg * momentum + mea
n * (1 - momentum)
            self.moving_var = self.moving_var * momentum + var
* (1 - momentum)
        else:
            mean = self.moving_avg
            var = self.moving_var

        x_norm = (x - mean) / (torch.sqrt(var + self.eps))
        return x_norm * self.gamma + self.beta
```

Two things I would like to highlight:

- We have different behaviour during training and during inference. On training we keep track of an exponential moving average of the mean and the variance, for later use during inference. The reason for this that we can obtain a much better estimation of mean and variance of the input over time while processing the batches during training, then use it on inference. It will be less accurate to use the mean and variance of the input batch during inference as likely the size is much smaller than what you used during training, the law of large numbers is playing a role here.
- And something I didn't know is that the layer contains a fully connected layer at the end (given by gamma and beta). That gives the ability to fully un-normalize the input and recover the original value if the network decides that's the best option. More on this in the following video:



When and Where should we use BatchNorm?

It seems that nearly always helps, so there isn't reason to not use it (unless the case on the next point). Usually it appears between a fully connected layer / conv layer and an activation layer. But also some people defend it's better to put it after the activations layer. I couldn't find any paper about using it after the activation, so the safest option is to do what everyone does, to put it before the activations.

When doesn't work?

When the samples of the batch are pretty similar, so similar that the mean/variance is basically 0, probably isn't a good idea to use BatchNorm.

Or in the extreme case of batches of size 1, it's just impossible to use it.

Tips and tricks

BatchNorm after conv layers

I have seen more than once that we shouldn't use bias on convolutional layers if we are using a BatchNorm after but I didn't know why, and I always forget about it.

Remember in the last step of BatchNorm we are multiplying and adding a number, like we do for any linear layer. It already adds it's own bias, and that's the reason I believe. Here a question on stack overflow about it.

Transfer learning

As we know an already trained network contains the moving average and variance of the dataset used to train it and this can be a problem. During transfer learning we typically freeze most of the layers, and if we aren't careful, BatchNorm layers too, meaning the moving averages applied belong to the original dataset not the new dataset.

It's a good idea to unfreeze the BatchNorm layers contained within the frozen layers to allow the network to recalculate the moving averages for you own data.

Machine Learning   Batch Norm   Deep Learning   Pytorch   Transfer Learning