PROBLEMS WITH TRAINING DEEP NEURAL NETS

INTERNAL COVARIATE SHIFT

It is the change in the distribution of network activations due to the change in network parameters during training.As we know that the output of the first layer feeds into the second layer and that of the second layer feeds into the third, and so on. So when the parameters of a layer change distribution of inputs to subsequent layers also change.To improve the training, we seek to reduce the internal covariate shift. By fixing the distribution of the layer inputs x as the training progresses, we expect to improve the training speed. It has been long known that the network training converges faster if its inputs are whitened – i.e., linearly transformed to have zero means and unit variances, and decorrelated.
Internal Covariate shift slows down the deep neural network training by requiring:

- Lower learning rate
- Careful parameter initialization
- And this makes the network hard to train models with saturating nonlinearities.

VANISHING AND EXPLODING GRADIENTS

Consider a layer with a sigmoid activation function z = g(Wu + b) where u is the layer input, the weight matrix W and bias vector b are the layer parameters to be learned, and g(x) = 1/(1+exp(−x)) . As |x| increases, g ′ (x) tends to zero. This means that for all dimensions of x = Wu+b except those with small absolute values, the gradient flowing down to u will vanish and the model will train slowly. However, since x is affected by W, b and the parameters of all the layers below, changes to those parameters during training will likely move many dimensions of x into the saturated regime of the nonlinearity and slow down the convergence. This effect is amplified as the network depth increases.
Saturating nonlinearities (like tanh or sigmoid) can not be used for deep networks as they tend to get stuck in the saturation region as the network grows deeper. Some ways around this are to use:

- Nonlinearities like ReLU which do not saturate
- Smaller learning rates
- Careful initialization

Now consider a layer with the input u that adds the learned bias b, and normalizes the result by subtracting the mean of the activation computed over the training data: $\widehat{x}$ = x − E[x] where x = u + b, X = {x1...N } is the set of values of x over the training set, and E[x] = 1/ N $\sum_{i=1}^{N} x_i$. If

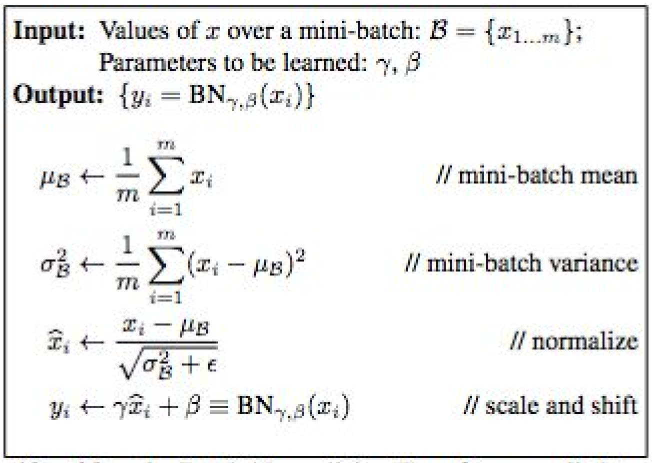
a gradient descent step ignores the dependence of E[x] on b, then it will update b ← b + Δb, where Δb ∝ −∂ℓ/∂xb. Then u + (b + Δb) − E[u + (b + Δb)] = u + b − E[u + b]. Thus, the combination of the update to b and subsequent change in normalization led to no change in the output of the layer nor, consequently, the loss. As the training continues, b will grow indefinitely while the loss remains fixed. This problem can get worse if the normalization not only centers but also scales the activations.So now ,
$\widehat{x}$ = Norm(x, X)
which depends not only on the given training example x but on all examples X
For backpropagation, we would need to compute the Jacobians
∂Norm(x, X)/∂x and ∂Norm(x, X)/ ∂X .ignoring the latter term would lead to the explosion described above.

NORMALIZATION

But within this framework described above, whitening the layer inputs is expensive, as it requires computing the covariance matrix Cov[x] = Ex∈X [xxT] − E[x]E[x]T and its inverse square root, to produce the whitened activations Cov[x]−1/2 (x − E[x]), as well as the derivatives of these transforms for backpropagation.So we use a different method.
Let us say that the layer we want to normalize has d dimensions x = (x1, ... xd). Then, we can normalize the kth dimension as follows:

$$y^k = \gamma^k \hat{x} + \beta^k$$

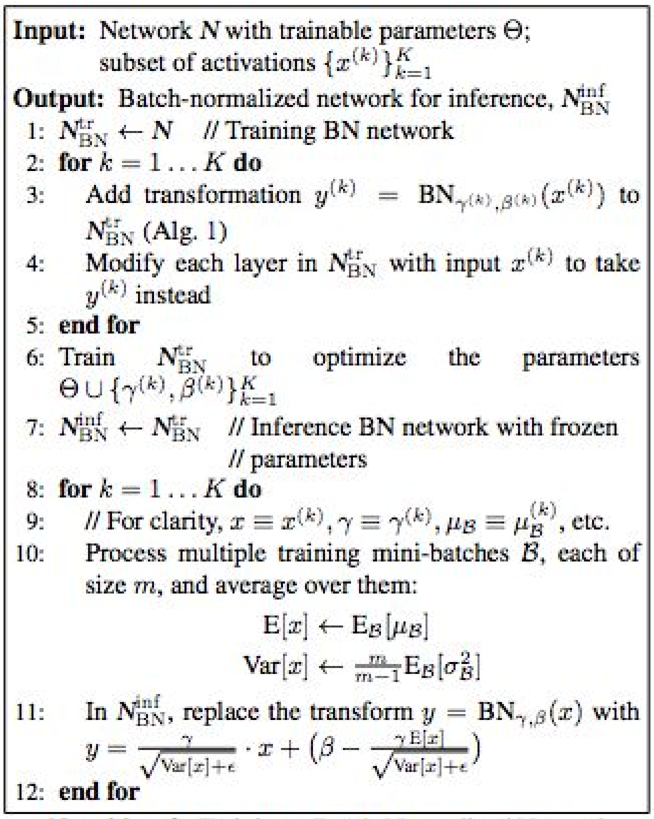
where γ and β are parameters to be learned.we start learning by setting $\gamma\ k = \sqrt{Var[x\ k\ ]}$ and $\beta\ k$ = E[$x\ k$].
Moreover, just like we use mini-batch in Stochastic Gradient Descent (SGD), we can use mini-batch with normalization to estimate the mean and variance for each activation.
The transformation from x to y as described above is called Batch Normalizing Transform. This BN transform is differentiable and ensures that as the model is training, the layers can learn on the input distributions that exhibit less internal covariate shift and can hence accelerate the training.
At training time, a subset of activations in specified and BN transform is applied to all of them.
During test time, the normalization is done using the population statistics instead of mini-batch statistics to ensure that the output deterministically depends on the input. Let’s understand the same using maths below.



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$    // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2$    // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$    // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$    // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.



```
Input: Network $N$ with trainable parameters $\Theta$;
       subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\mathrm{BN}}^{\mathrm{inf}}$
1: $N_{\mathrm{BN}}^{\mathrm{tr}} \leftarrow N$    // Training BN network
2: for $k = 1 \dots K$ do
3:     Add transformation $y^{(k)} = \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\mathrm{BN}}^{\mathrm{tr}}$ (Alg. 1)
4:     Modify each layer in $N_{\mathrm{BN}}^{\mathrm{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: end for
6: Train $N_{\mathrm{BN}}^{\mathrm{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\mathrm{BN}}^{\mathrm{inf}} \leftarrow N_{\mathrm{BN}}^{\mathrm{tr}}$    // Inference BN network with frozen
                                // parameters
8: for $k = 1 \dots K$ do
9:     // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10:    Process multiple training mini-batches $\mathcal{B}$, each of size $m$, and average over them:
           $\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$
           $\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$
11:    In $N_{\mathrm{BN}}^{\mathrm{inf}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$
12: end for
```

Algorithm 2: Training a Batch-Normalized Network

Inference here means testing.

BATCH NORMALIZED CONVOLUTIONAL NETWORKS

Let us say that x = g(Wu+b) is the operation performed by the layer where W and b are the parameters to be learned, g is a nonlinearity and u is the input from the previous layer.
The BN transform is added just before the nonlinearity, by normalizing x = Wu+b. An alternative would have been to normalize u itself but constraining just the first and the second moment would not eliminate the covariate shift from u.
When normalizing Wu+b, we can ignore the b term as it would be canceled during the normalization step (b's role is subsumed by β) and we have
z = g(BN(Wu))
For convolutional layers, normalization should follow the convolution property as well - i.e. different elements of the same feature map, at different locations, are normalized in the same way. So all the activations in a mini-batch are jointly normalized over all the locations and parameters (γ and β) are learnt per feature map instead of per activation.
For convolutions, every layer/filter/kernel is normalized on its own (linear layer: each neuron/node/component). That means that every generated value ("pixel") is treated as an example. If we have a batch size of N and the image generated by the convolution has

width=P and height=Q, we would calculate the mean (E) over N*P*Q examples (same for the variance).

ACCELERATING BATCH NORMALIZED NETWORKS

Simply adding Batch Normalization to a network does not take full advantage .To do so, we further changed the network and its training parameters, as follows:

- **Increase learning rate**

  We can use large learning rate to increase training speed with batch normalized model ,because of less danger of exploding/vanishing gradients

- **Remove Dropout And Reduce L2 weight Regularization**

BN reduces demand for regularization, e.g. dropout or L2 norm. (Because the means and variances are calculated over batches and therefore every normalized value depends on the current batch. I.e. the network can no longer just memorize values and their correct answers.)

- **Remove Local Response Normalization**

While Inception and other networks like Alexnet benefit from it, experiments found that with Batch Normalization it is not necessary.

- **Shuffle Training examples more thoroughly**
- **Reduce the photometric Distortions**

Because batch normalized networks train faster and observe each training example fewer times, we let the trainer focus on more “real” images by distorting them less.

CONCLUSION

BN is similar to a normalization layer suggested by Gülcehre and Bengio. However, they applied it to the outputs of nonlinearities. They also didn't have the beta and gamma parameters (i.e. their normalization could not learn the identity function).
Merely using Batch Normalization speeds up convergence of the network and allows better results over a single network.
Applying other modification (higher learning rates, removing /decreasing dropout etc.)that are affordable by batch Normalization reaches former state of the art in a fraction of steps