CS60021: Scalable Data Mining

Large Scale Machine Learning

Sourangshu Bhattacharya

BATCH NORMALIZATION

Slides taken from Jude Shavlik: http://pages.cs.wisc.edu/~shavlik/cs638_cs838.html

Motivation

- The range of values of raw training data often varies widely
  - Example: Has kids feature in {0,1}
  - Value of car: $500-$100'sk

- In machine learning algorithms, the functions involved in the optimization process are sensitive to normalization
  - For example: Distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature.
  - After, normalization, each feature contributes approximately proportionately to the final distance.

- In general, Gradient descent converges much faster with feature scaling than without it.
- Good practice for numerical stability for numerical calculations, and to avoid ill-conditioning when solving systems of equations.

Common normalizations

Two methods are usually used for rescaling or normalizing data:

- Scaling data all numeric variables to the range [0,1]. One possible formula is given below:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- To have zero mean and unit variance:

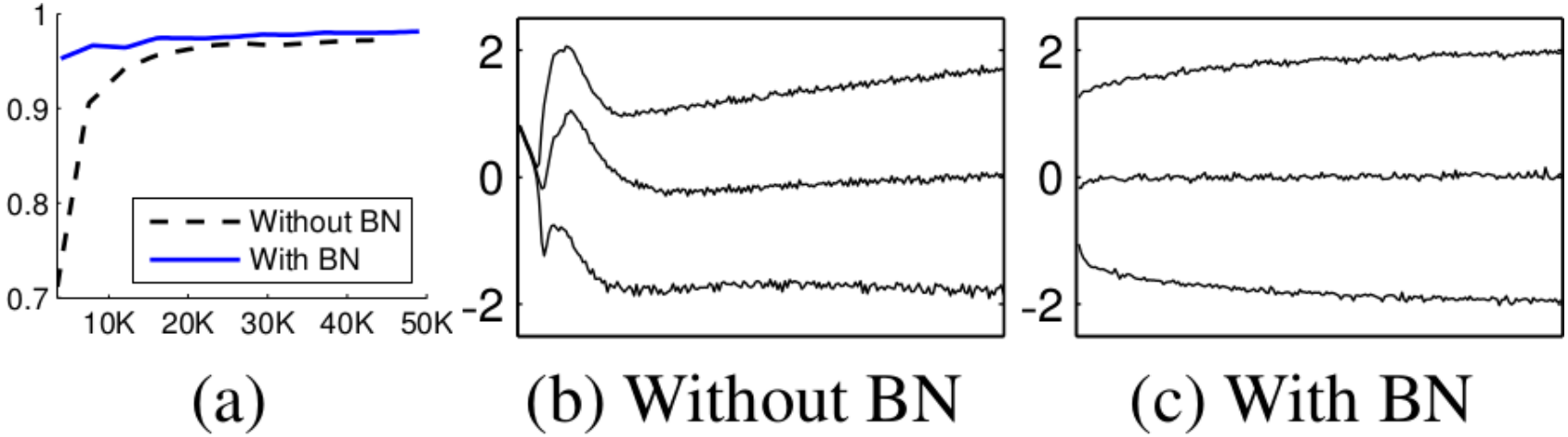
$$x_{new} = \frac{x - \mu}{\sigma}$$

- In the NN community this is call *Whitening*

Batch normalization: Other benefits in practice

- BN reduces training times. (Because of less Covariate Shift, less exploding/vanishing gradients.)
- 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.)
- BN allows higher learning rates. (Because of less danger of exploding/vanishing gradients.)
- BN enables training with saturating nonlinearities in deep networks, e.g. sigmoid. (Because the normalization prevents them from getting stuck in saturating ranges, e.g. very high/low values for sigmoid.)

Batch normalization: Better accuracy , faster.



BN applied to MNIST (a), and activations of a randomly selected neuron over time (b, c), where the middle line is the median activation, the top line is the 15th percentile and the bottom line is the 85th percentile.

Why the naïve approach Does not work?

- Normalizes layer inputs to zero mean and unit variance. *whitening*.
- Naive method: Train on a batch. Update model parameters. Then normalize. Doesn't work: Leads to exploding biases while distribution parameters (mean, variance) don't change.
  - If we do it this way gradient always ignores the effect that the normalization for the next batch would have
  - i.e. : "The issue with the above approach is that the gradient descent optimization does not take into account the fact that the normalization takes place"

Doing it the “correct way” is too expensive!

- A proper method has to include the current example batch *and* somehow all previous batches (all examples) in the normalization step.

- This leads to calculating in covariance matrix and its inverse square root. That's expensive. The authors found a faster way!

The issue with the above approach is that the gradient descent optimization does not take into account the fact that the normalization takes place. To address this issue, we would like to ensure that, for any parameter values, the network *always* produces activations with the desired distribution. Doing so would allow the gradient of the loss with respect to the model parameters to account for the normalization, and for its dependence on the model parameters Θ. Let again x be a layer input, treated as a vector, and $\mathcal{X}$ be the set of these inputs over the training data set. The normalization can then be written as a transformation

$$\hat{x} = \text{Norm}(x, \mathcal{X})$$

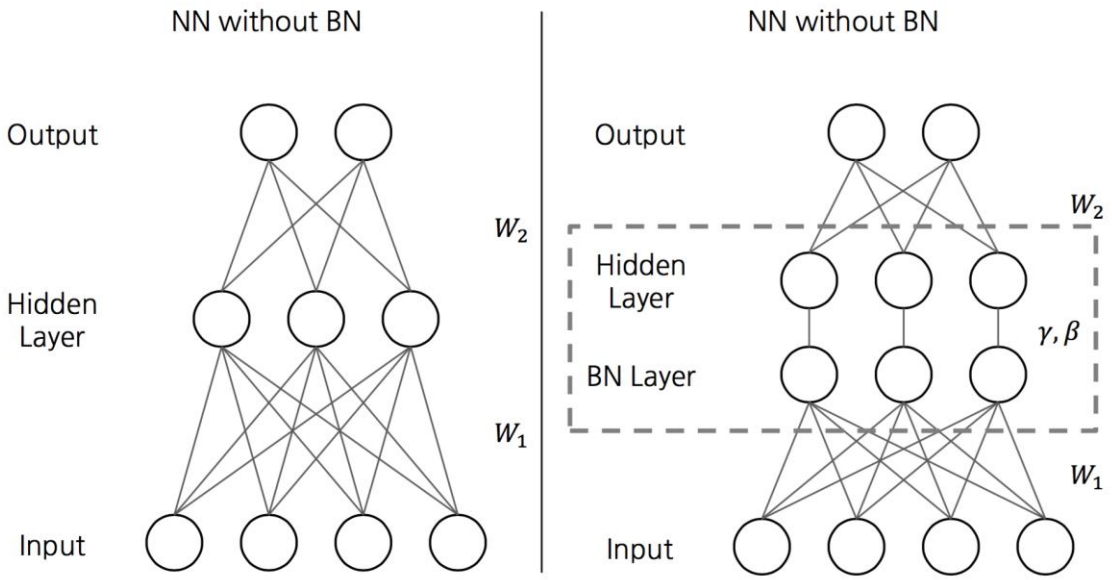
which depends not only on the given training example x but on all examples $\mathcal{X}$ – each of which depends on Θ if x is generated by another layer. For backpropagation, we would need to compute the Jacobians $\frac{\partial \text{Norm}(x, \mathcal{X})}{\partial x}$ and $\frac{\partial \text{Norm}(x, \mathcal{X})}{\partial \mathcal{X}}$; ignoring the latter term would lead to the explosion described above. Within this framework, whitening the layer inputs is expensive, as it requires computing the covariance matrix $\text{Cov}[x] = \text{E}_{x \in \mathcal{X}}[xx^T] - \text{E}[x]\text{E}[x]^T$ and its inverse square root, to produce the whitened activations $\text{Cov}[x]^{-1/2}(x - \text{E}[x])$, as well as the derivatives of these transforms for backpropagation. This motivates us to seek an alternative that performs input normalization in a way that is differentiable and does not require the analysis of the entire training set after every parameter update.

The proposed solution:

To add an extra regularization layer

we introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}, \beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}.$$



A new layer is added so the gradient can “see” the normalization and make adjustments if needed.

Algorithm Summary: Normalization via Mini-Batch Statistics

- Each feature (component) is normalized individually
- Normalization according to:
  - componentNormalizedValue = (componentOldValue - E[component]) / sqrt(Var(component))
- A new layer is added so the gradient can "see" the normalization and made adjustments if needed.
  - The new layer has the power to learn the identity function to de-normalize the features if necessary!
  - Full formula: newValue = gamma * componentNormalizedValue + beta (gamma and beta learned per component)
- E and Var are estimated for each mini batch.
- BN is fully differentiable.

The Batch Transformation: formally from the paper.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

The full algorithm as proposed in the paper

Input: Network N with trainable parameters Θ; subset of activations $\{x^{(k)}\}_{k=1}^{K}$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network

2: **for** $k = 1 \ldots K$ **do**

3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)

4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead

5: **end for**

6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$

7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters

8: **for** $k = 1 \ldots K$ **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:

$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$

12: **end for**

Algorithm 2: Training a Batch-Normalized Network

Alg 1 (previous slide)

Architecture modification

Note that BN(x) is different during test…

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

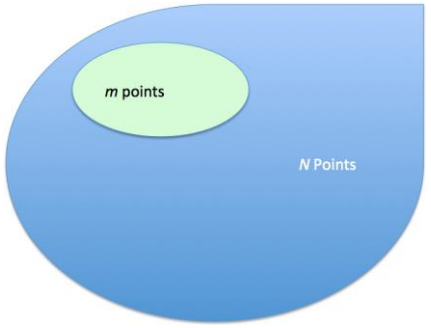
Vs.

$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

Populations stats vs. sample stats

- In algorithm 1, we are estimating the true mean and variance over the entire population for a given batch.

- When doing inference you're minibatching your way through the entire dataset, you're calculating statistics on a per sample/batch basis. We want our sample statistics to be *unbiased* to population statistics.



	Population Statistics over *N*	Sample/Batch Statistics over *m*
Mean Estimate	$\mu = \frac{1}{N}\sum_i x_i$	$\bar{x} = \frac{1}{m}\sum_i x_i$
Variance Estimate	$\sigma = \frac{1}{N}\sum_i (x_i - \mu)^2$	$\sigma_B = \frac{1}{m-1}\sum_i (x_i - \bar{x})^2$

ACCELERATING BN NETWORKS
Batch normalization only not enough!

- Increase learning rate.
- Remove Dropout.
- Shuffle training examples more thoroughly
- Reduce the L2 weight regularization.
- Accelerate the learning rate decay.
- Reduce the photometric distortions.

References:

- Ioffe, Sergey, and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In *International Conference on Machine Learning*, pp. 448-456. 2015.

- http://pages.cs.wisc.edu/~shavlik/cs638_cs838.html