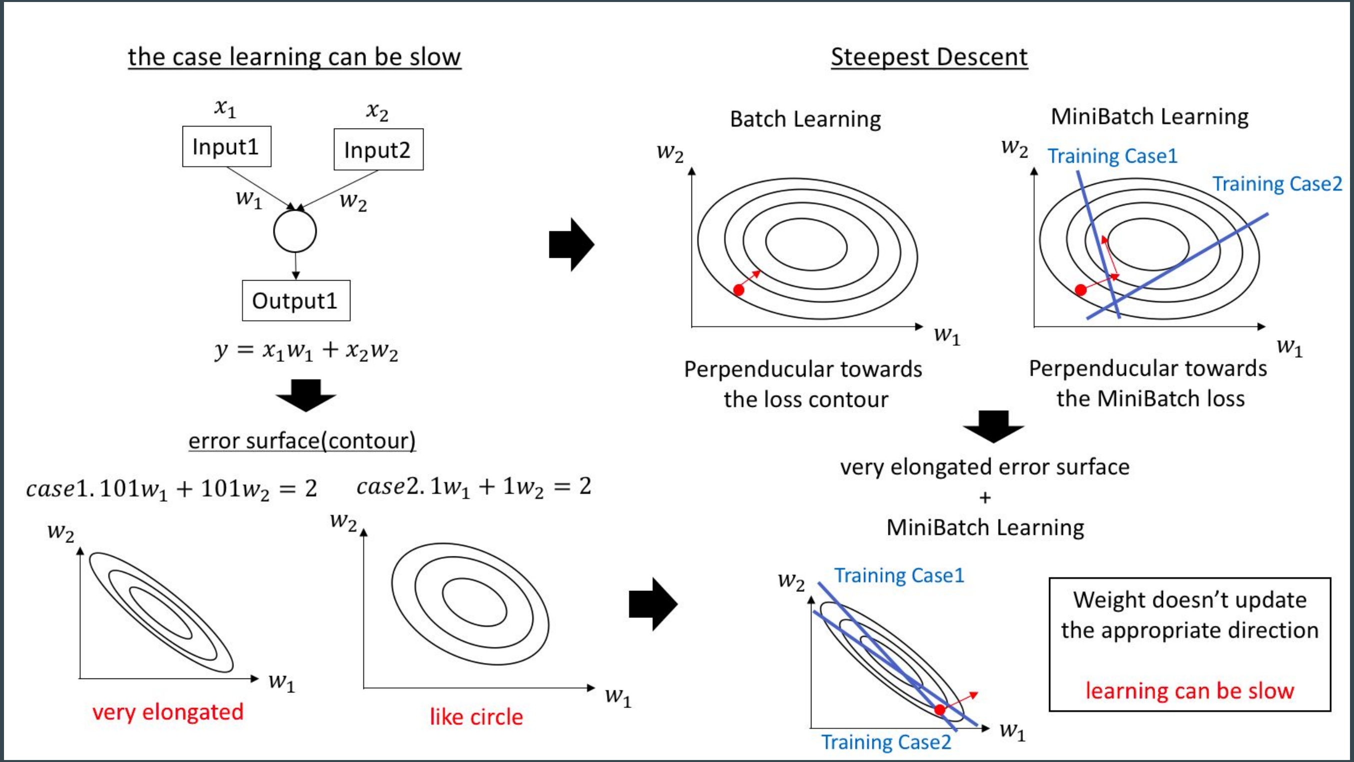
Normalization

•••

Sepehr Sameni (@Separius)

Normalization & Standardization



Batch Normalization

- Formula + Inference
- Internal Covariate Shift
- Normalization
- Batch Size
- Smoothness => higher LR
- Batch norm location
- Generalization
- Recurrent Batch Norm
- Conditional Batch Norm

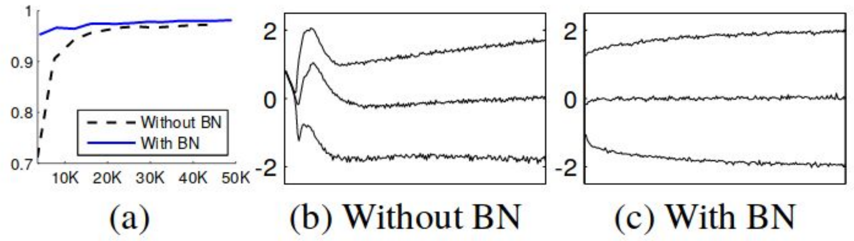
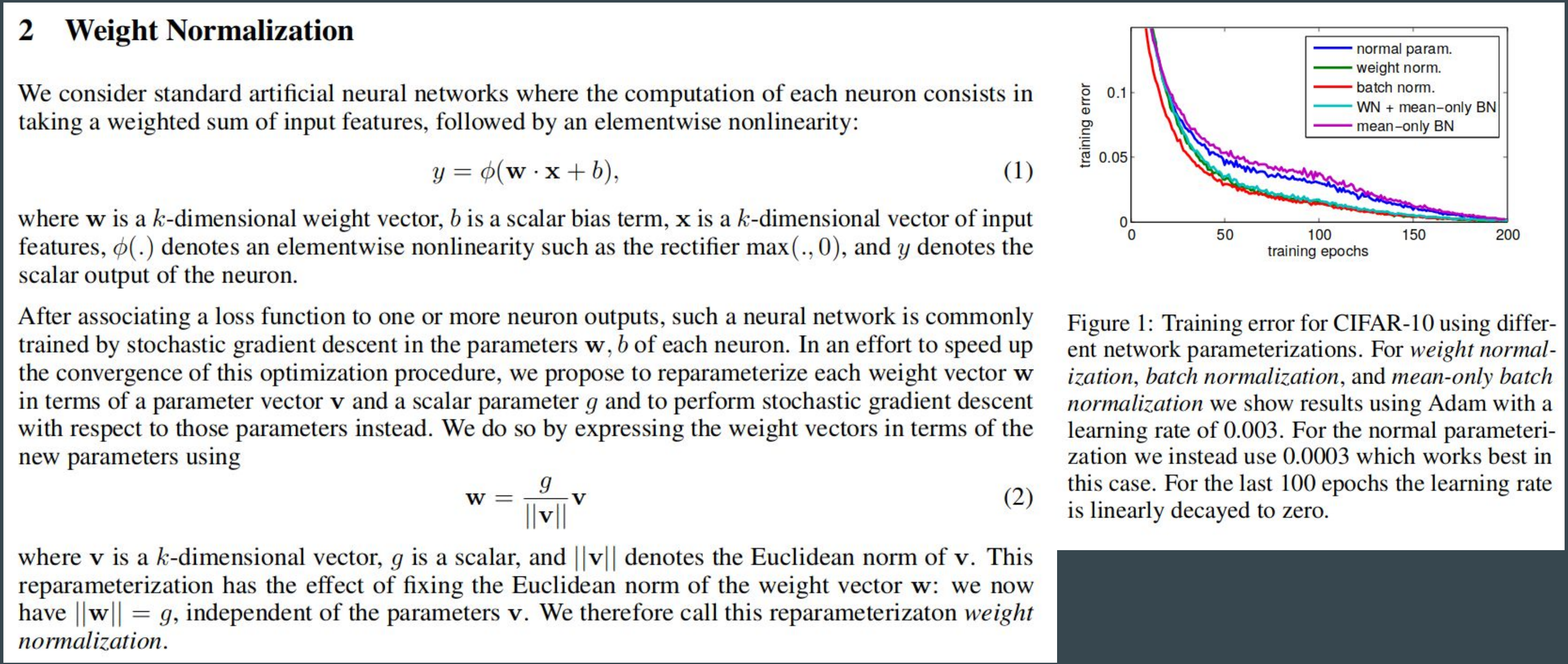


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Weight Norm



2 Weight Normalization

We consider standard artificial neural networks where the computation of each neuron consists in taking a weighted sum of input features, followed by an elementwise nonlinearity:

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b), \tag{1}$$

where $\mathbf{w}$ is a k-dimensional weight vector, b is a scalar bias term, $\mathbf{x}$ is a k-dimensional vector of input features, $\phi(.)$ denotes an elementwise nonlinearity such as the rectifier $\max(., 0)$, and y denotes the scalar output of the neuron.

After associating a loss function to one or more neuron outputs, such a neural network is commonly trained by stochastic gradient descent in the parameters $\mathbf{w}, b$ of each neuron. In an effort to speed up the convergence of this optimization procedure, we propose to reparameterize each weight vector $\mathbf{w}$ in terms of a parameter vector $\mathbf{v}$ and a scalar parameter g and to perform stochastic gradient descent with respect to those parameters instead. We do so by expressing the weight vectors in terms of the new parameters using

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v} \tag{2}$$

where $\mathbf{v}$ is a k-dimensional vector, g is a scalar, and $||\mathbf{v}||$ denotes the Euclidean norm of $\mathbf{v}$. This reparameterization has the effect of fixing the Euclidean norm of the weight vector $\mathbf{w}$: we now have $||\mathbf{w}|| = g$, independent of the parameters $\mathbf{v}$. We therefore call this reparameterizaton *weight normalization*.

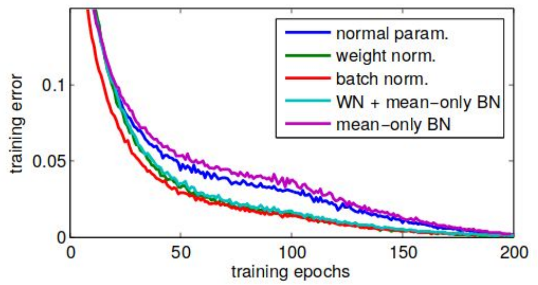
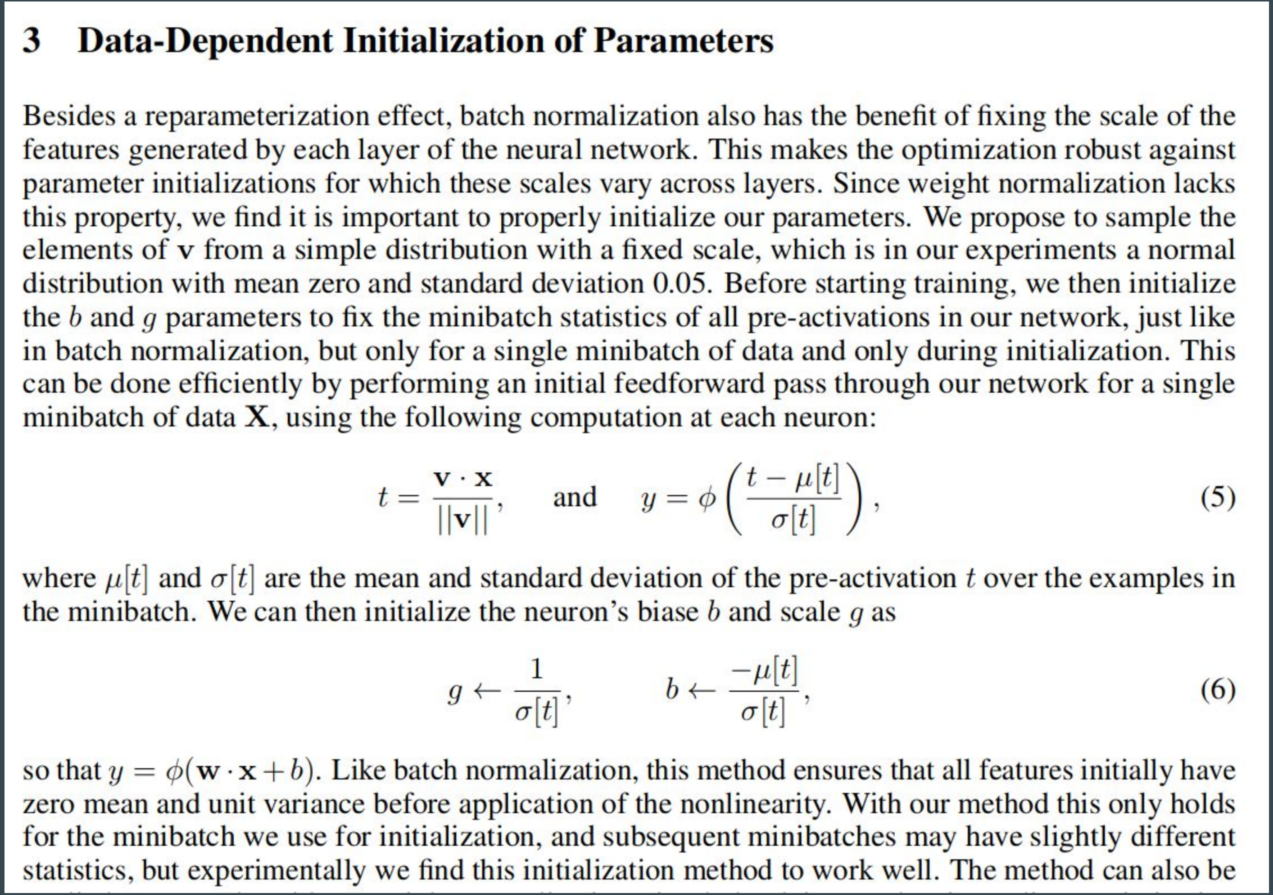


Figure 1: Training error for CIFAR-10 using different network parameterizations. For *weight normalization*, *batch normalization*, and *mean-only batch normalization* we show results using Adam with a learning rate of 0.003. For the normal parameterization we instead use 0.0003 which works best in this case. For the last 100 epochs the learning rate is linearly decayed to zero.

Weight Norm.init



3 Data-Dependent Initialization of Parameters

Besides a reparameterization effect, batch normalization also has the benefit of fixing the scale of the features generated by each layer of the neural network. This makes the optimization robust against parameter initializations for which these scales vary across layers. Since weight normalization lacks this property, we find it is important to properly initialize our parameters. We propose to sample the elements of $\mathbf{v}$ from a simple distribution with a fixed scale, which is in our experiments a normal distribution with mean zero and standard deviation 0.05. Before starting training, we then initialize the b and g parameters to fix the minibatch statistics of all pre-activations in our network, just like in batch normalization, but only for a single minibatch of data and only during initialization. This can be done efficiently by performing an initial feedforward pass through our network for a single minibatch of data $\mathbf{X}$, using the following computation at each neuron:

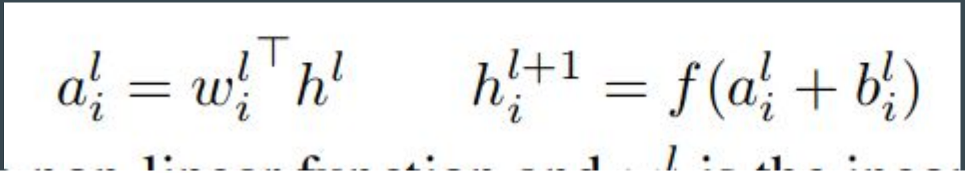
$$t = \frac{\mathbf{v} \cdot \mathbf{x}}{||\mathbf{v}||}, \quad \text{and} \quad y = \phi\left(\frac{t - \mu[t]}{\sigma[t]}\right), \tag{5}$$

where $\mu[t]$ and $\sigma[t]$ are the mean and standard deviation of the pre-activation t over the examples in the minibatch. We can then initialize the neuron's biase b and scale g as

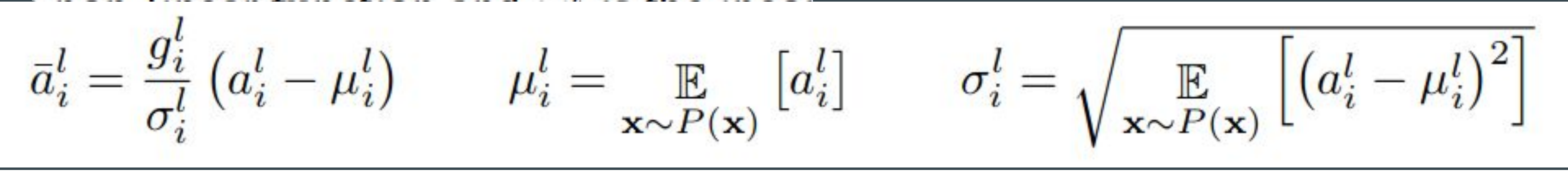
$$g \leftarrow \frac{1}{\sigma[t]}, \qquad b \leftarrow \frac{-\mu[t]}{\sigma[t]}, \tag{6}$$

so that $y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$. Like batch normalization, this method ensures that all features initially have zero mean and unit variance before application of the nonlinearity. With our method this only holds for the minibatch we use for initialization, and subsequent minibatches may have slightly different statistics, but experimentally we find this initialization method to work well. The method can also be

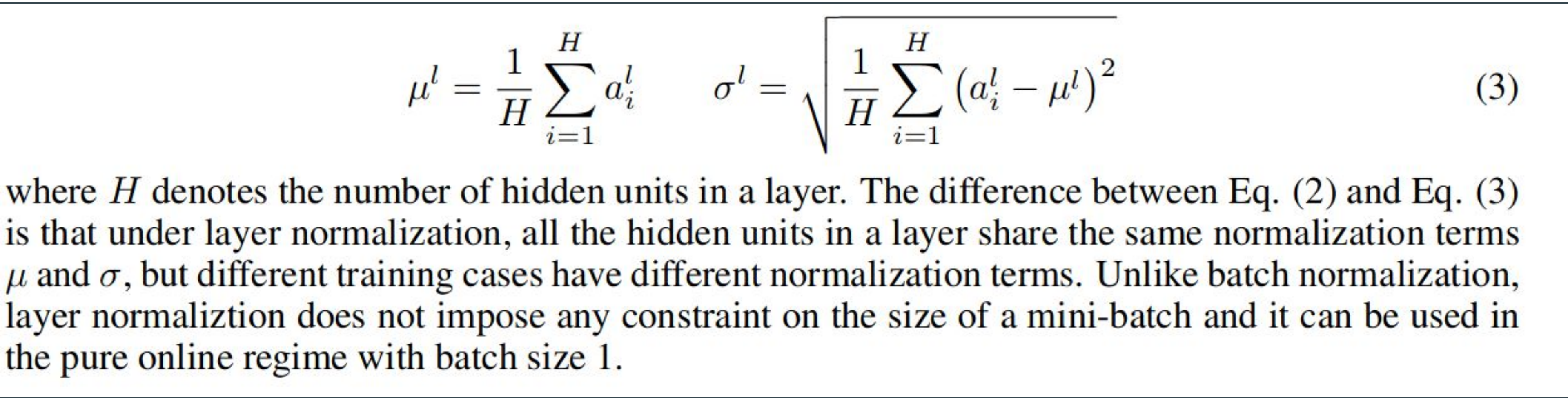
LayerNorm



$$a_i^l = {w_i^l}^\top h^l \qquad h_i^{l+1} = f(a_i^l + b_i^l)$$



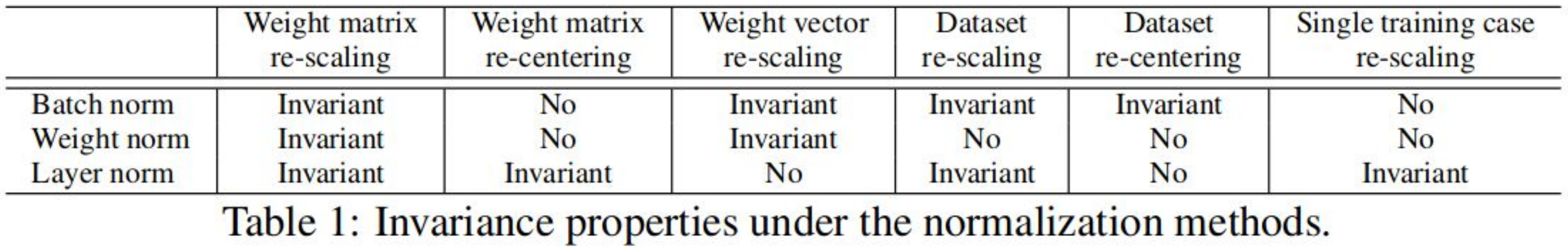
$$\bar{a}_i^l = \frac{g_i^l}{\sigma_i^l}\left(a_i^l - \mu_i^l\right) \qquad \mu_i^l = \mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[a_i^l\right] \qquad \sigma_i^l = \sqrt{\mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[\left(a_i^l - \mu_i^l\right)^2\right]}$$



$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \tag{3}$$

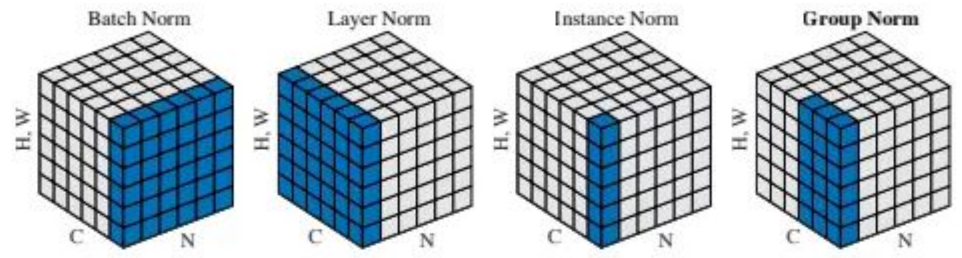
where H denotes the number of hidden units in a layer. The difference between Eq. (2) and Eq. (3) is that under layer normalization, all the hidden units in a layer share the same normalization terms μ and σ, but different training cases have different normalization terms. Unlike batch normalization, layer normaliztion does not impose any constraint on the size of a mini-batch and it can be used in the pure online regime with batch size 1.

Summary

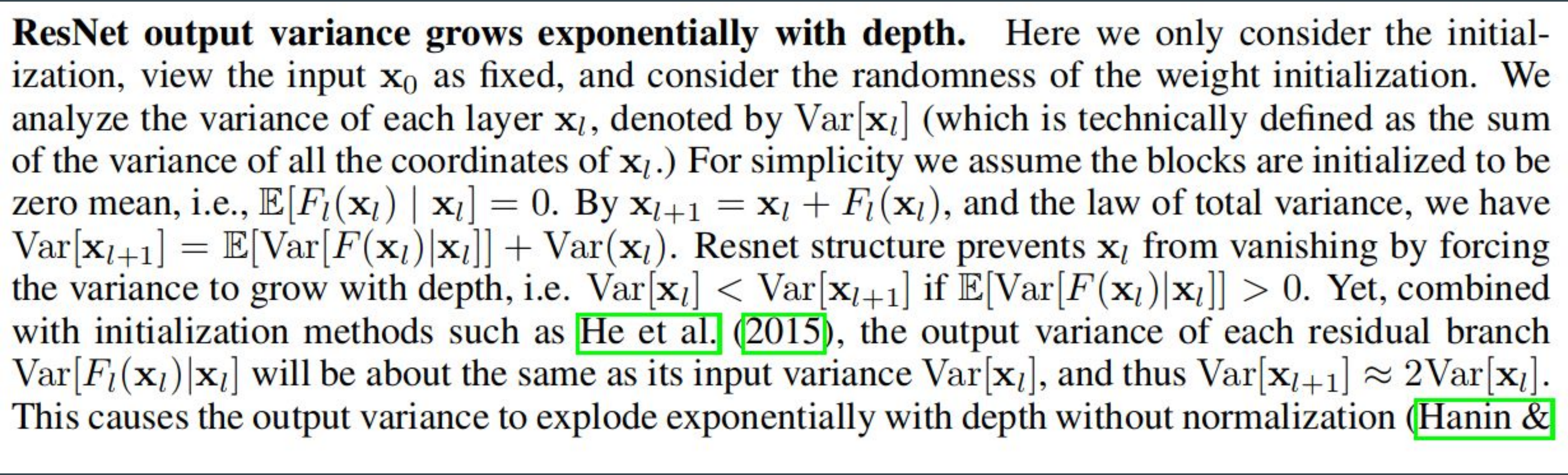


	Weight matrix re-scaling	Weight matrix re-centering	Weight vector re-scaling	Dataset re-scaling	Dataset re-centering	Single training case re-scaling
Batch norm	Invariant	No	Invariant	Invariant	Invariant	No
Weight norm	Invariant	No	Invariant	No	No	No
Layer norm	Invariant	Invariant	No	Invariant	No	Invariant

Table 1: Invariance properties under the normalization methods.



FixUp(problem)



ResNet output variance grows exponentially with depth. Here we only consider the initialization, view the input $\mathbf{x}_0$ as fixed, and consider the randomness of the weight initialization. We analyze the variance of each layer $\mathbf{x}_l$, denoted by $\mathrm{Var}[\mathbf{x}_l]$ (which is technically defined as the sum of the variance of all the coordinates of $\mathbf{x}_l$.) For simplicity we assume the blocks are initialized to be zero mean, i.e., $\mathbb{E}[F_l(\mathbf{x}_l) \mid \mathbf{x}_l] = 0$. By $\mathbf{x}_{l+1} = \mathbf{x}_l + F_l(\mathbf{x}_l)$, and the law of total variance, we have $\mathrm{Var}[\mathbf{x}_{l+1}] = \mathbb{E}[\mathrm{Var}[F(\mathbf{x}_l)|\mathbf{x}_l]] + \mathrm{Var}(\mathbf{x}_l)$. Resnet structure prevents $\mathbf{x}_l$ from vanishing by forcing the variance to grow with depth, i.e. $\mathrm{Var}[\mathbf{x}_l] < \mathrm{Var}[\mathbf{x}_{l+1}]$ if $\mathbb{E}[\mathrm{Var}[F(\mathbf{x}_l)|\mathbf{x}_l]] > 0$. Yet, combined with initialization methods such as He et al. (2015), the output variance of each residual branch $\mathrm{Var}[F_l(\mathbf{x}_l)|\mathbf{x}_l]$ will be about the same as its input variance $\mathrm{Var}[\mathbf{x}_l]$, and thus $\mathrm{Var}[\mathbf{x}_{l+1}] \approx 2\mathrm{Var}[\mathbf{x}_l]$. This causes the output variance to explode exponentially with depth without normalization (Hanin &

FixUp(solution)

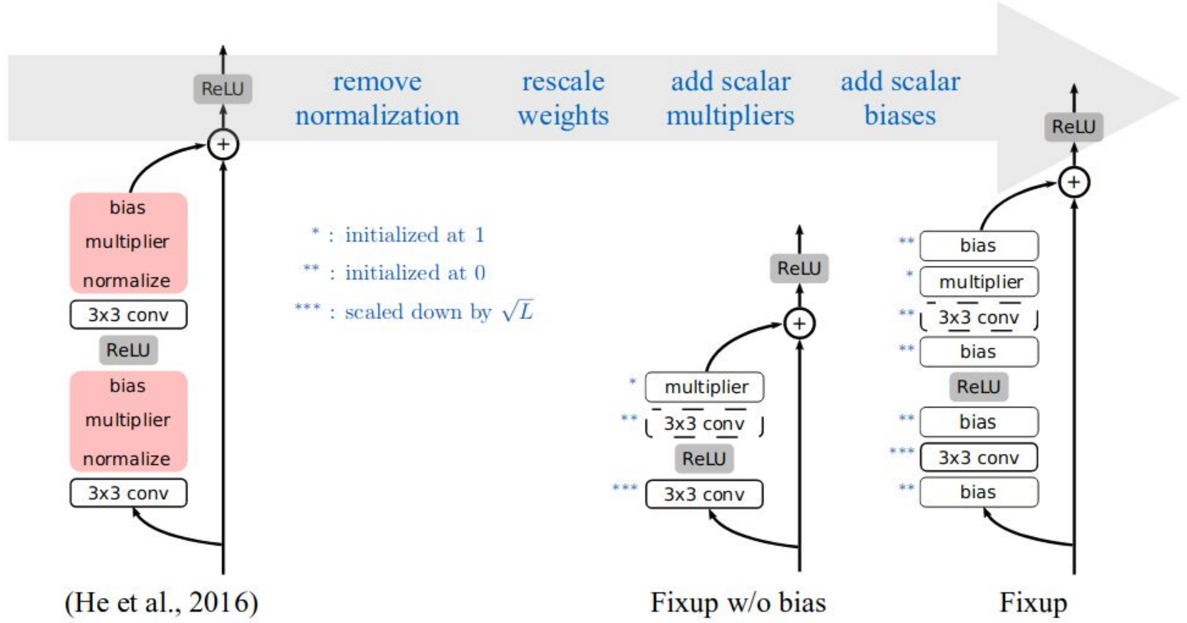
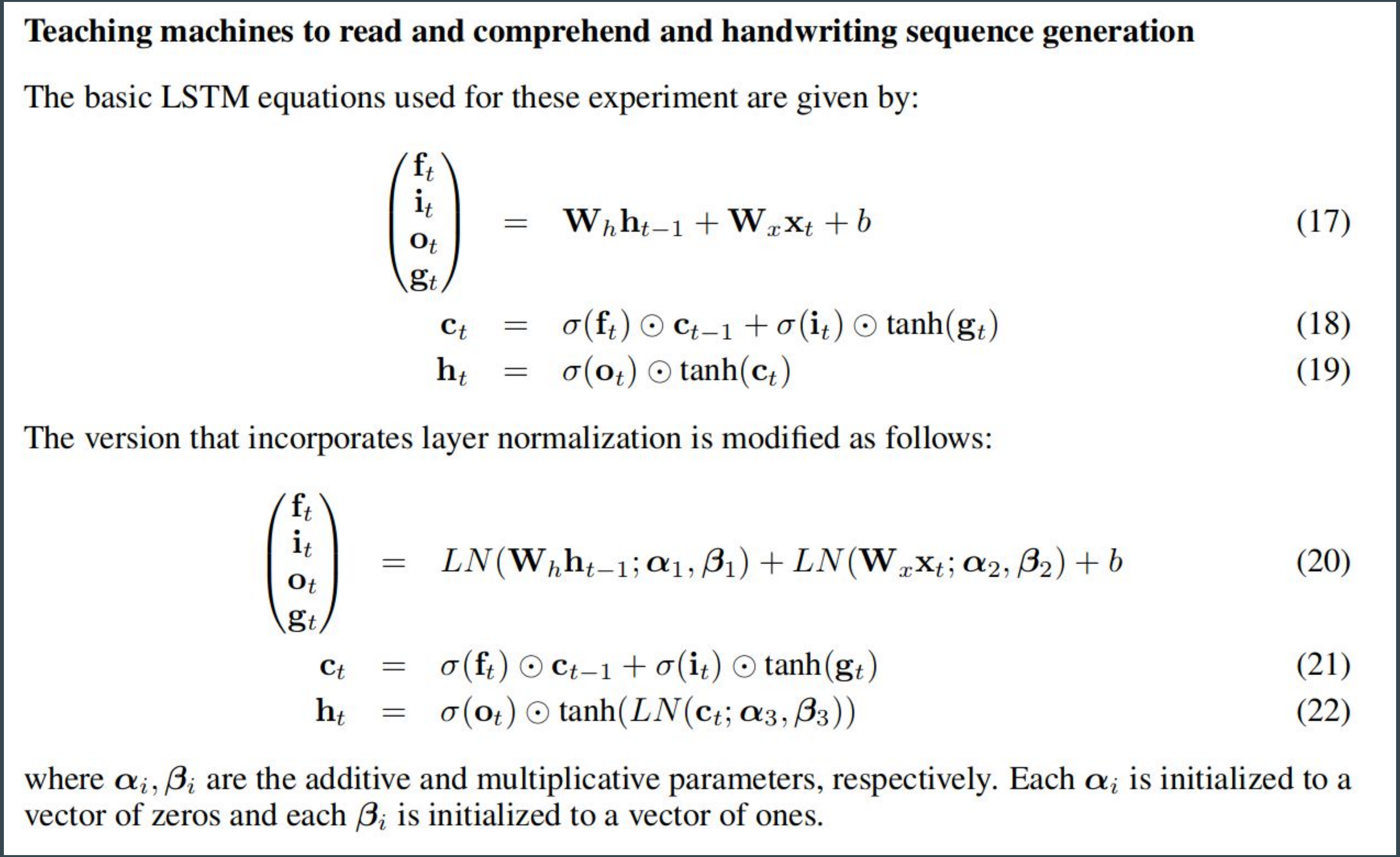


Figure 1: **Left:** ResNet basic block. Batch normalization (Ioffe & Szegedy, 2015) layers are marked in red. **Middle:** A simple network block that trains stably when stacked together. **Right:** Fixup further improves by adding bias parameters. (See Section 3 for details.)

Backup slide



Teaching machines to read and comprehend and handwriting sequence generation

The basic LSTM equations used for these experiment are given by:

$$
\begin{aligned}
\begin{pmatrix} \mathbf{f}_t \\ \mathbf{i}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} &= \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + b && (17) \\
\mathbf{c}_t &= \sigma(\mathbf{f}_t) \odot \mathbf{c}_{t-1} + \sigma(\mathbf{i}_t) \odot \tanh(\mathbf{g}_t) && (18) \\
\mathbf{h}_t &= \sigma(\mathbf{o}_t) \odot \tanh(\mathbf{c}_t) && (19)
\end{aligned}
$$

The version that incorporates layer normalization is modified as follows:

$$
\begin{aligned}
\begin{pmatrix} \mathbf{f}_t \\ \mathbf{i}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} &= LN(\mathbf{W}_h \mathbf{h}_{t-1}; \boldsymbol{\alpha}_1, \boldsymbol{\beta}_1) + LN(\mathbf{W}_x \mathbf{x}_t; \boldsymbol{\alpha}_2, \boldsymbol{\beta}_2) + b && (20) \\
\mathbf{c}_t &= \sigma(\mathbf{f}_t) \odot \mathbf{c}_{t-1} + \sigma(\mathbf{i}_t) \odot \tanh(\mathbf{g}_t) && (21) \\
\mathbf{h}_t &= \sigma(\mathbf{o}_t) \odot \tanh(LN(\mathbf{c}_t; \boldsymbol{\alpha}_3, \boldsymbol{\beta}_3)) && (22)
\end{aligned}
$$

where $\boldsymbol{\alpha}_i, \boldsymbol{\beta}_i$ are the additive and multiplicative parameters, respectively. Each $\boldsymbol{\alpha}_i$ is initialized to a vector of zeros and each $\boldsymbol{\beta}_i$ is initialized to a vector of ones.