Optimization for Training I

First-Order Methods Training algorithm

Dinghuai Zhang (slides by Prof. Aaron Courville)

OPTIMIZATION METHODS

Topics: Types of optimization methods.

- Practical optimization methods breakdown into two categories:
  1. First-order methods
  2. Second-order methods

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{a}) + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{a})(\boldsymbol{\theta} - \boldsymbol{a}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{a})^{\top} \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{a})$$

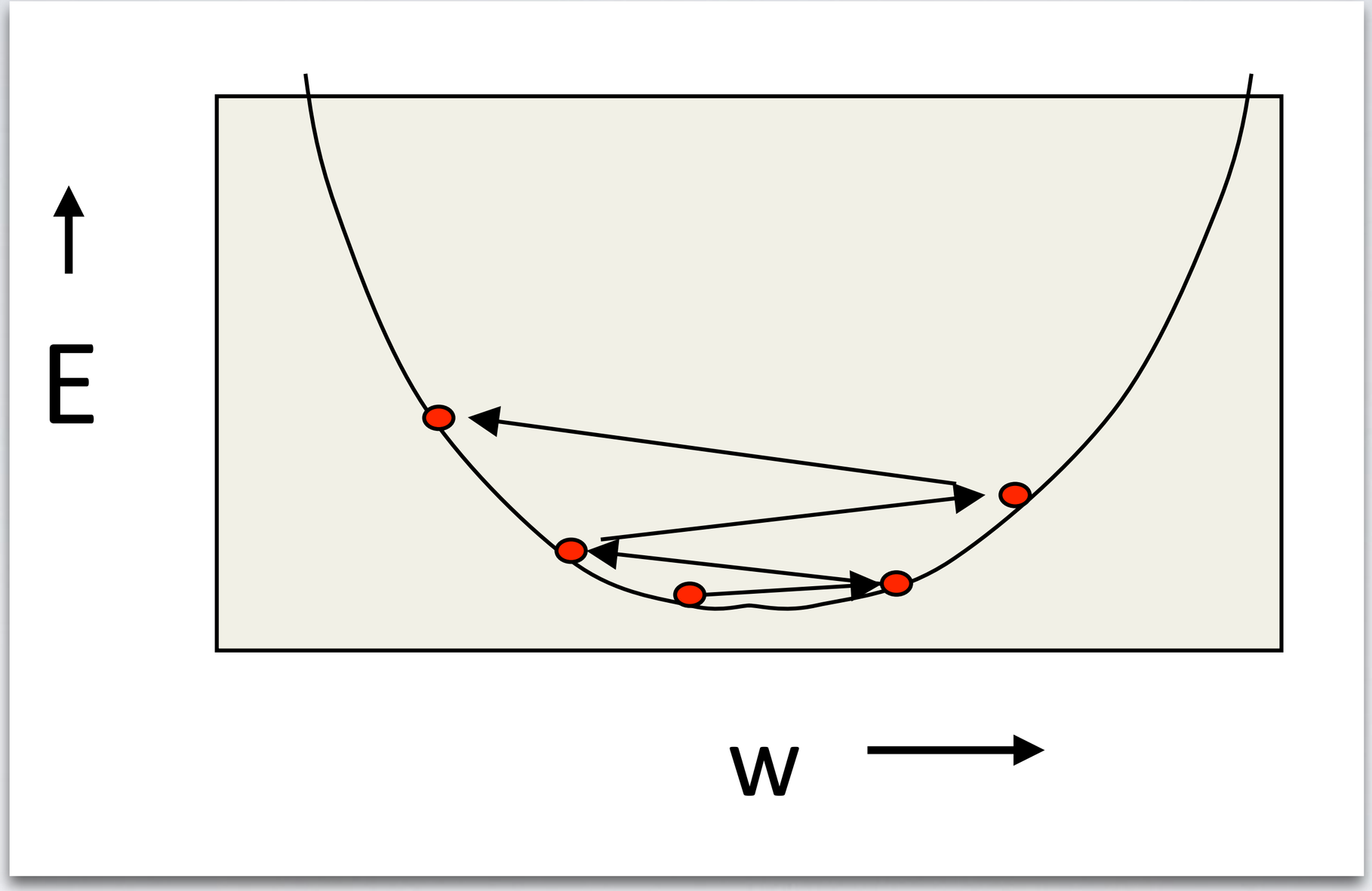
STOCHASTIC GRADIENT DESCENT

- Vanilla SGD is still a popular method of training deep learning models.
- (+) Works on a single example or a mini-batch / (-) Can converge slowly.

Algorithm 1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{h}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{h}}$
  end while

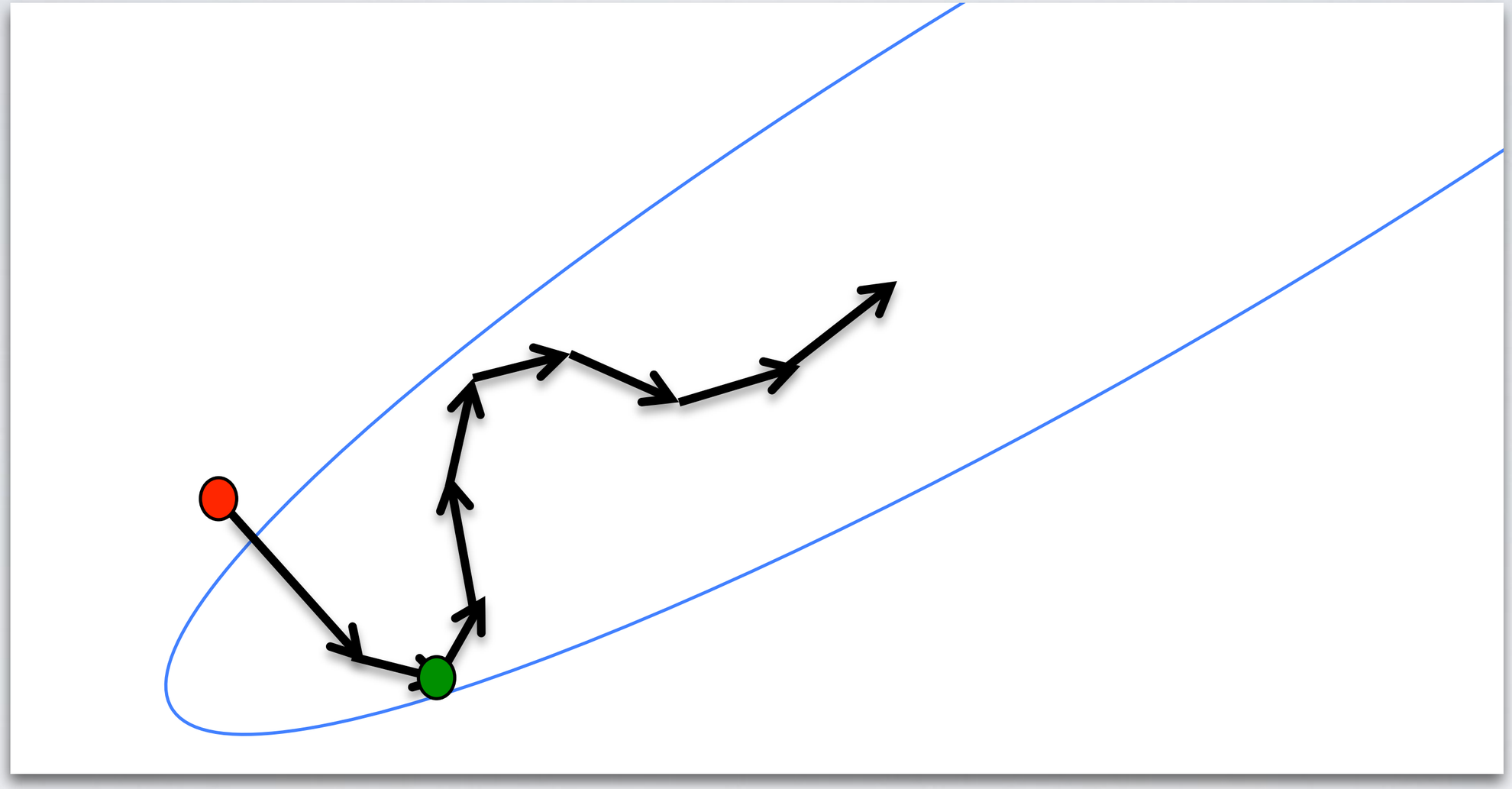
STOCHASTIC GRADIENT DESCENT



MOMENTUM METHOD

- Designed to accelerate learning, especially with small consistent gradients.
- Inspired from physical interpretation of the optimization process: Imagine you have a small ball rolling on a surface defined by the loss function.

MOMENTUM METHOD



MOMENTUM METHOD

Algorithm 1 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{h} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{h}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

NESTEROV MOMENTUM

- Sutskever et al (ICML 2013) presented a modified version of momentum they called Nesterov momentum.
- Basic idea: apply the gradient "correction" after the velocity term is applied.

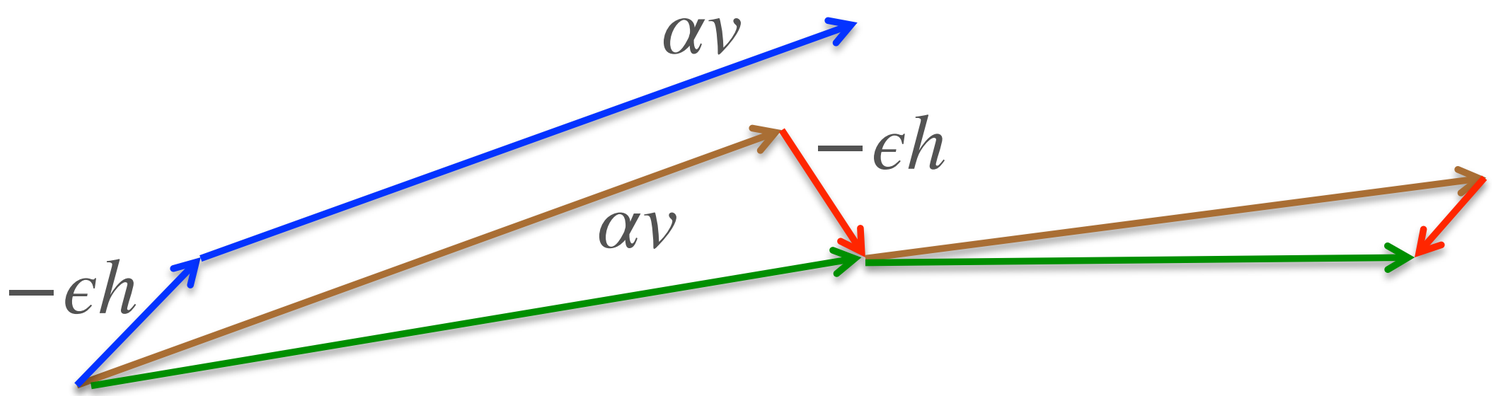
NESTEROV MOMENTUM

Algorithm 1 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient (at interim point): $\boldsymbol{h} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{h}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

NESTEROV MOMENTUM

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Slide from Hinton's Coursera course.

ADAGRAD

- Adagrad (Duchi et al, COLT 2010) is a method of adapting the learning rate.

- (+) Can adapt independent learning rates for all parameters
- (-) Accumulating gradients from the start makes later learning very slow.

Great for convex optimization. Less so for neural network optimization.

ADAGRAD

Algorithm 1 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{h} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{h} \odot \boldsymbol{h}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{h}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSPROP

- Modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.
- Compared to AdaGrad, the use of the moving average introduces a new hyperparameter that controls the length scale of the moving average.
- Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks, though it can be difficult to tune properly.

RMSPROP

Algorithm 1 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{h} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{h} \odot \boldsymbol{h}$

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{h}$. $(\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied elem-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

RMSPROP+MOMENTUM

Algorithm 1 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.

Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$

Compute gradient: $\boldsymbol{h} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{h} \odot \boldsymbol{h}$

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{h}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

ADAM

- ``Adam" derives from the phrase ``adaptive moments."
- Variant of RMSProp + momentum with a few important distinctions:
  1. Momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient.
  2. Includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.
- To date, Adam has largely become the default optimization algorithm for training deep learning systems.

ADAM:

Algorithm 1 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization. (Suggestion: 10^{-8})

Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{h} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{h}$

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{h} \odot \boldsymbol{h}$

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

AMSGRAD

Reddi et al. (ICLR 2018) ON THE CONVERGENCE OF ADAM AND BEYOND

- Observation: Adam fails to converge to an optimal solution (or a critical point in nonconvex settings) due to the exponential moving average.
- Idea: Add a "long-term memory" of past gradients to Adam.

Replace: $\Delta\boldsymbol{\theta} = -\dfrac{\epsilon}{\sqrt{\hat{\boldsymbol{r}}} + \delta}\hat{\boldsymbol{s}}$

with: $\Delta\boldsymbol{\theta} = -\dfrac{\epsilon}{\sqrt{\hat{\boldsymbol{r}}^{\max}} + \delta}\hat{\boldsymbol{s}}$ where $\hat{\boldsymbol{r}}^{\max} = \max\left(\hat{\boldsymbol{r}}^{\max}, \hat{\boldsymbol{r}}\right)$

Initialize: $\hat{\boldsymbol{r}}^{\max} = 0$

ADABOUND

Luo et al. (ICLR 2019) ADAPTIVE GRADIENT METHODS WITH DYNAMIC BOUND OF LEARNING RATE

- Observation: Adam optimizes well but suffers poor generalization relative to SGD.
- Idea: Clip the effective learning rate in Adam to a "good" operating regime

Replace: $\Delta\boldsymbol{\theta} = -\dfrac{\epsilon}{\sqrt{\hat{\boldsymbol{r}}} + \delta}\hat{\boldsymbol{s}}$ **with:** $\Delta\boldsymbol{\theta} = -\,\mathrm{Clip}\left(\dfrac{\epsilon}{\sqrt{\hat{\boldsymbol{r}}} + \delta}, \eta_l, \eta_u\right)\hat{\boldsymbol{s}}$

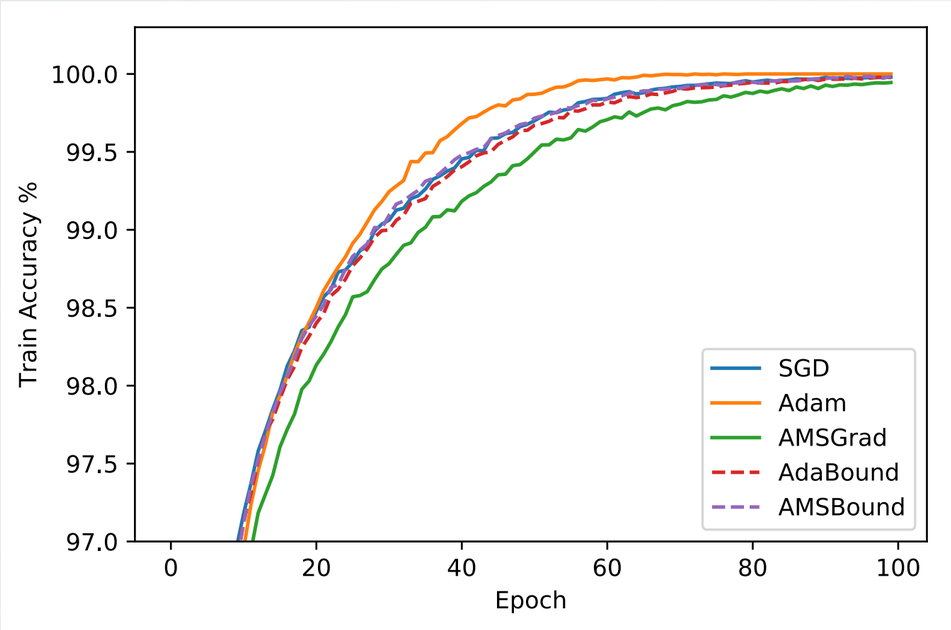
where (as suggested): $\eta_l = 0.1 - \dfrac{0.1}{(1-\rho_2)t + 1}, \qquad \eta_u = 0.1 + \dfrac{0.1}{(1-\rho_2)t}$

t in the the above eqns. represents update time.

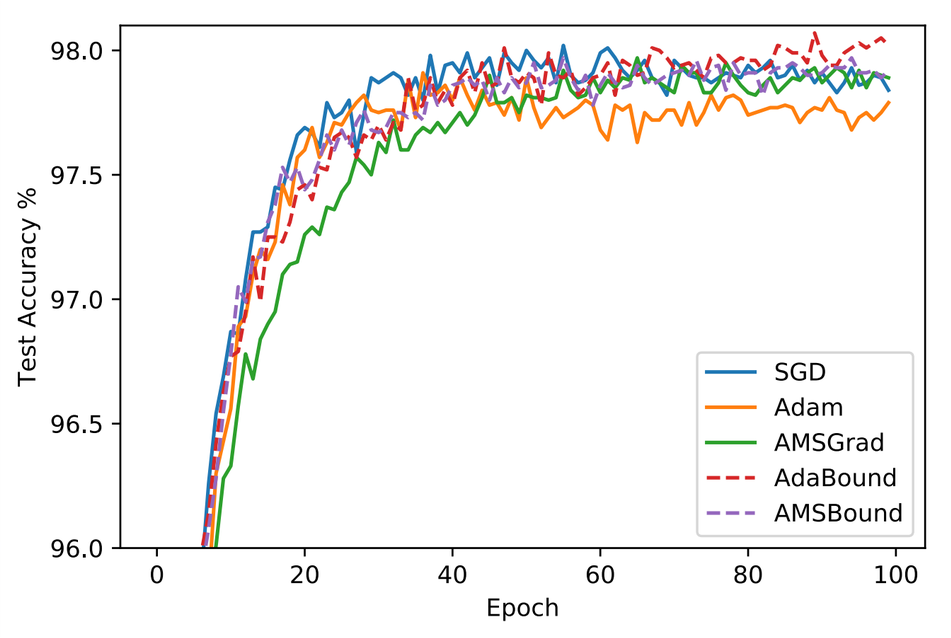
ADABOUND

Luo et al. (ICLR 2019) ADAPTIVE GRADIENT METHODS WITH DYNAMIC BOUND OF LEARNING RATE

- Observation: Adam optimizes well but suffers poor generalization relative to SGD.
- Idea: Clip the effective learning rate in Adam to a "good" operating regime



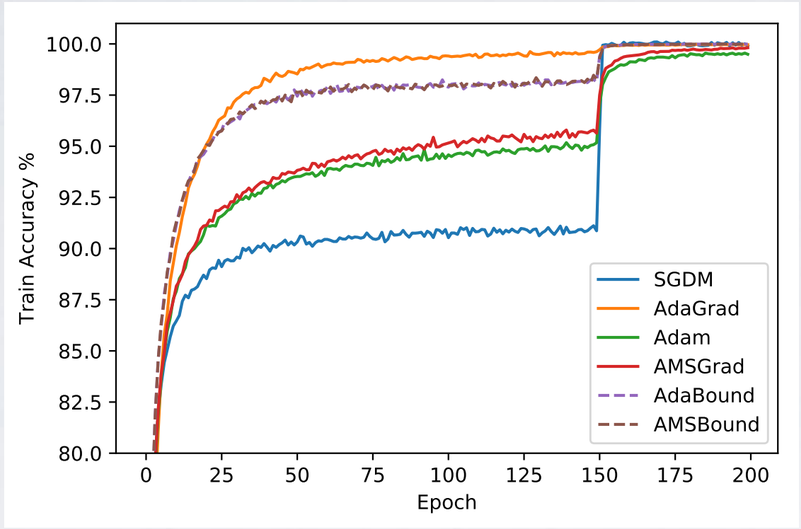
(a) Training Accuracy



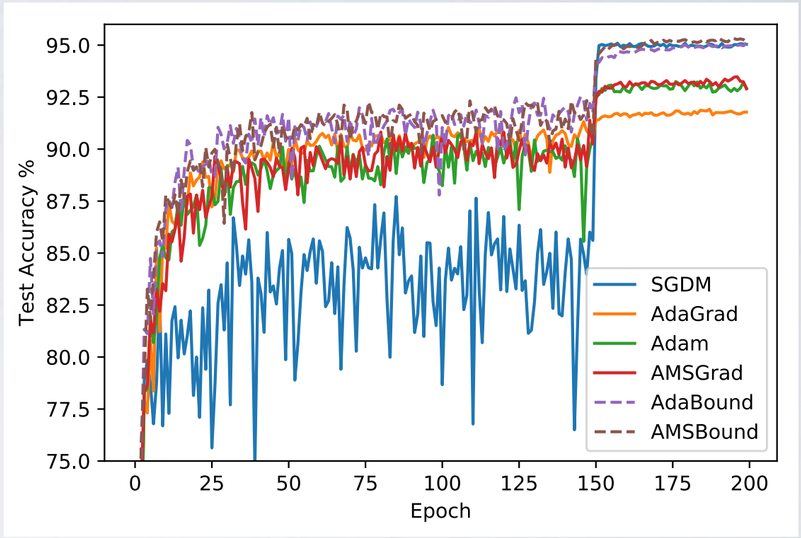
(b) Test Accuracy

ADABOUND

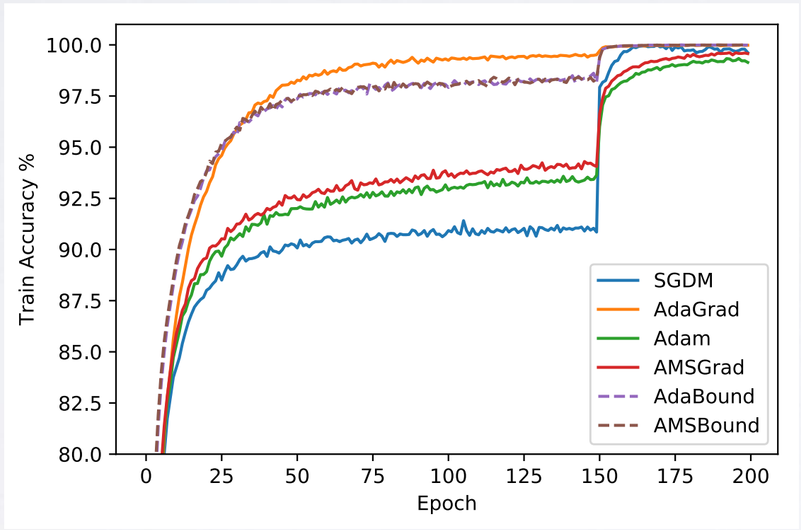
Image from: Luo et al. (ICLR 2019) ADAPTIVE GRADIENT METHODS WITH DYNAMIC BOUND OF LEARNING RATE



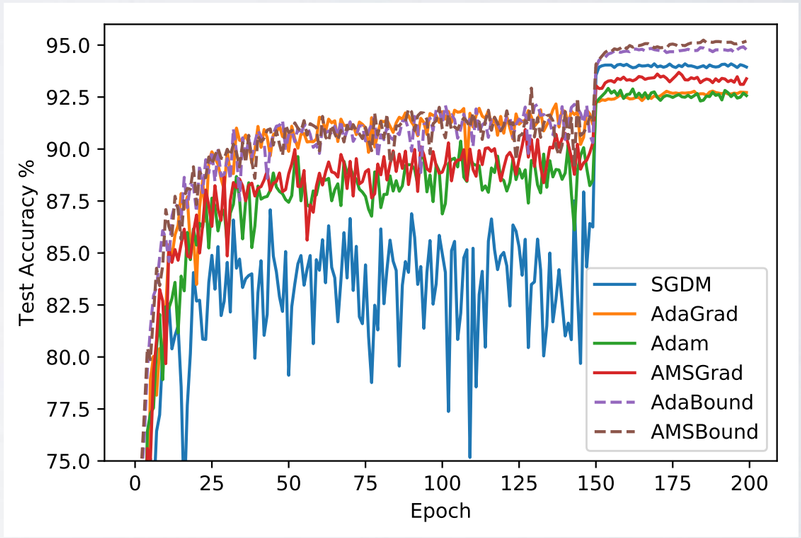
(a) Training Accuracy for DenseNet-121



(b) Test Accuracy for DenseNet-121



(c) Training Accuracy for ResNet-34



(d) Test Accuracy for ResNet-34

Note: learning_rate/10 @ epoch 150.

Figure 3: Training and test accuracy for DenseNet-121 and ResNet-34 on CIFAR-10.

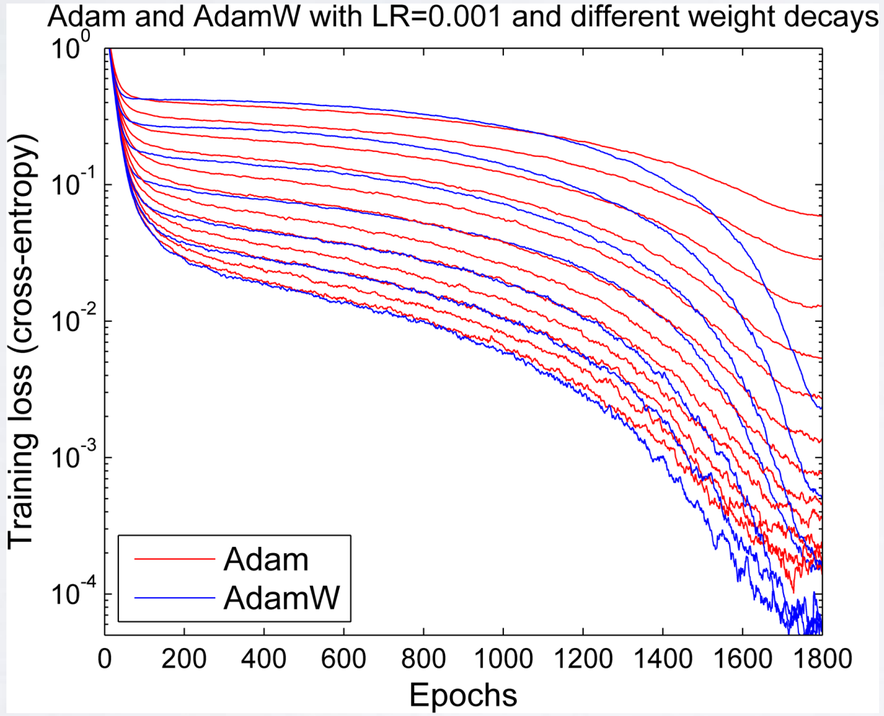
ADAMW

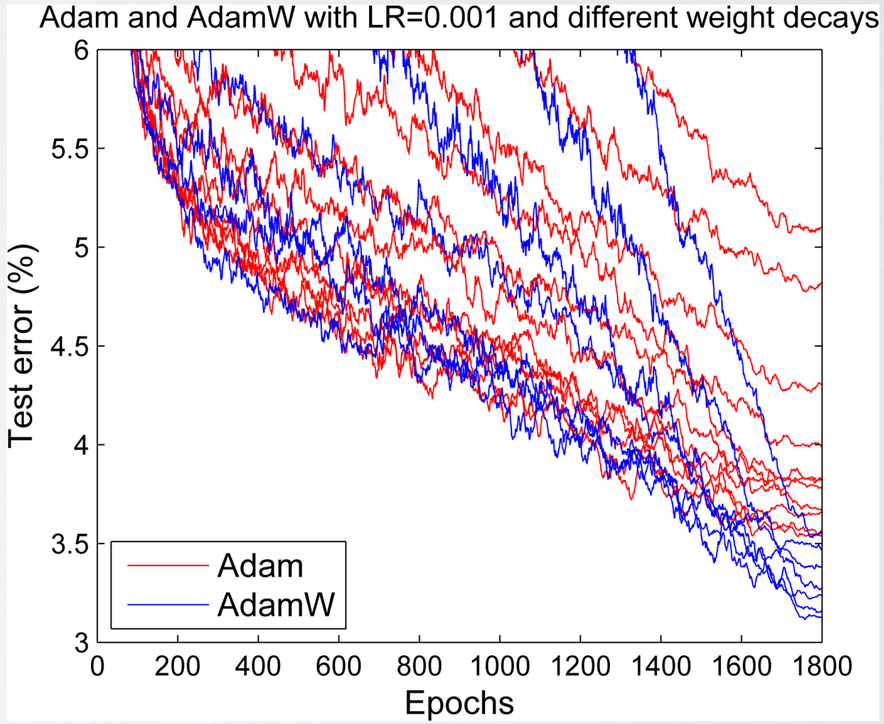
Loshchilov and Hutter (ICLR 2019) DECOUPLED WEIGHT DECAY REGULARIZATION

- Observation: Weight decay and L_2 regularization are only equivalent for SGD, not for Adam. Also L_2 regularization seems ineffective with Adam .

- Idea: Decouple weight decay from the other contributions to the gradient (loss).

ResNet trained on CIFAR-10





ADAMW:

Loshchilov and Hutter (ICLR 2019)
DECOUPLED WEIGHT DECAY REGULARIZATION

- Introduces weight decay as an independent process from the loss: WD not L2 regularization.
- Popular with Transformers and ConvNeXt

input : γ(lr), β_1, β_2(betas), θ_0(params), $f(\theta)$(objective), ϵ (epsilon)
λ(weight decay), *amsgrad*, *maximize*

initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** … **do**

if *maximize* :

$g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$

else

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (a)

$\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$ (b)

Weight decay introduced in (b) not (a)

$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$

$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

$\widehat{m_t} \leftarrow m_t / \big(1-\beta_1^t\big)$

$\widehat{v_t} \leftarrow v_t / \big(1-\beta_2^t\big)$

if *amsgrad*

$\widehat{v_t}^{max} \leftarrow \mathbf{max}(\widehat{v_t}^{max}, \widehat{v_t})$

$\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$

else

$\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

return θ_t

Source: PyTorch documentation
https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html

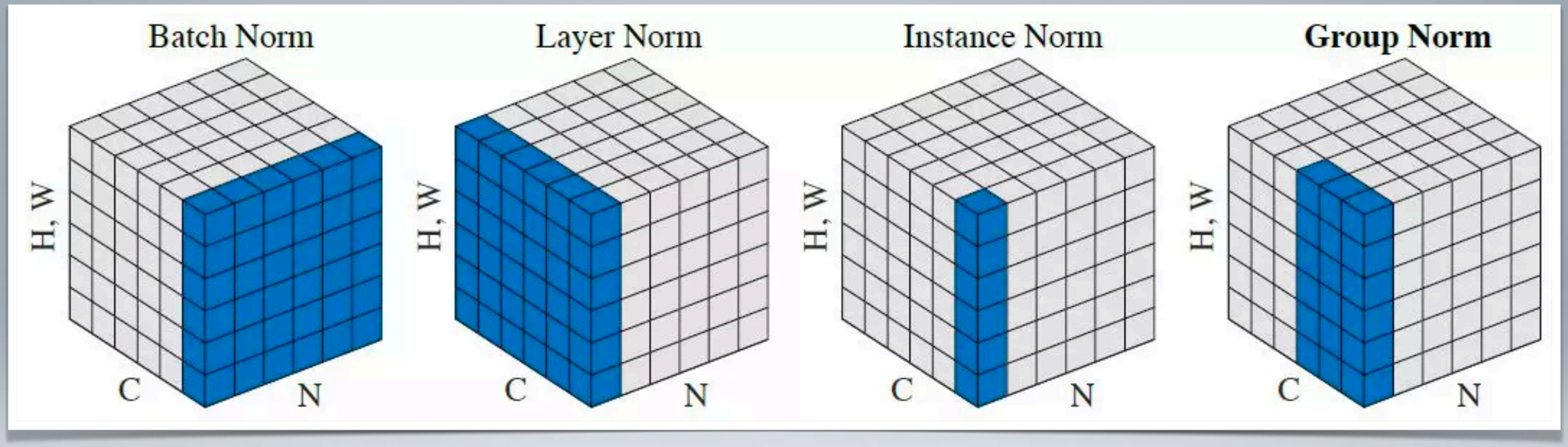
ADAMW:

Loshchilov and Hutter (ICLR 2019) DECOUPLED WEIGHT DECAY REGULARIZATION

Algorithm Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

Source: Original AdamW paper (citation above)



NORMALIZATION METHODS

BATCH NORMALIZATION

- S Ioffe, C Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, ICML 2015
- Reparametrization of the model with statistics computed across the minibatch.

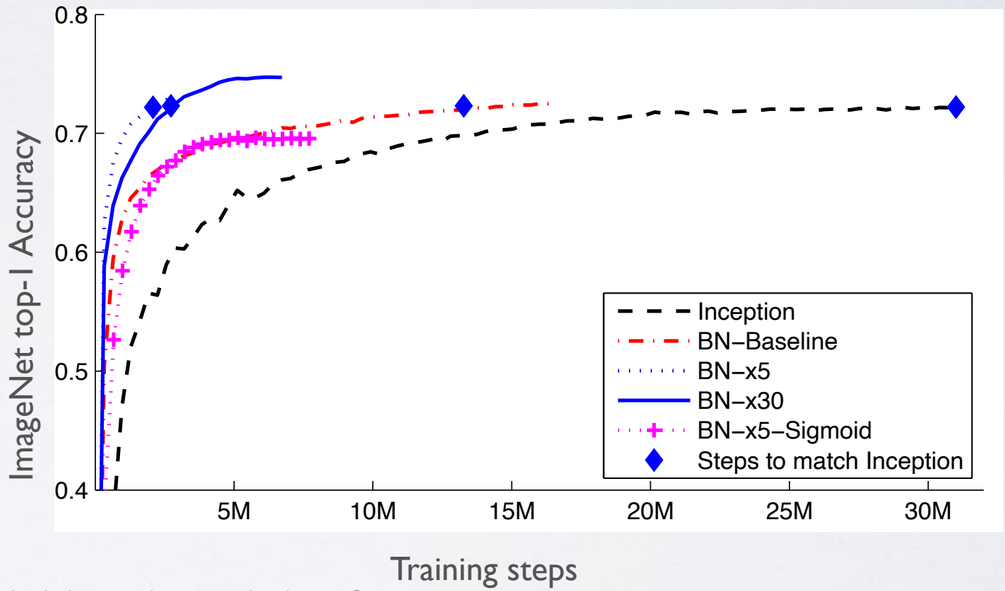
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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



- For CNNs the statistics are computed across both the mini-batch and the feature map.

H = height
W = width
T = examples in mini-batch

$$\mu_i = \frac{1}{HWT}\sum_{t=1}^{T}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm}, \qquad \sigma_i^2 = \frac{1}{HWT}\sum_{t=1}^{T}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm} - mu_i)^2.$$

LAYER NORMALIZATION

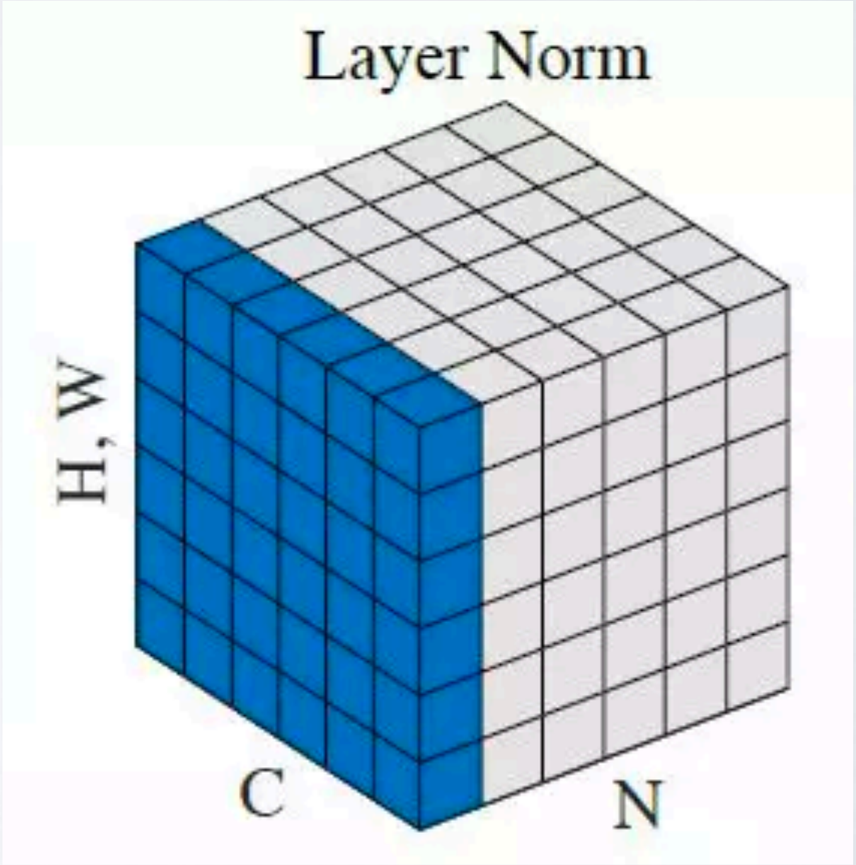
- Normalize units across the units in a layer (not across the mini-batch as in batchnorm)
- Standard normalization for RNNs and Transformers

$$y_i = \gamma \hat{x}_i + \beta$$

$$\hat{x}_i = \frac{x_i - \mu^l}{\sigma^l}$$

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

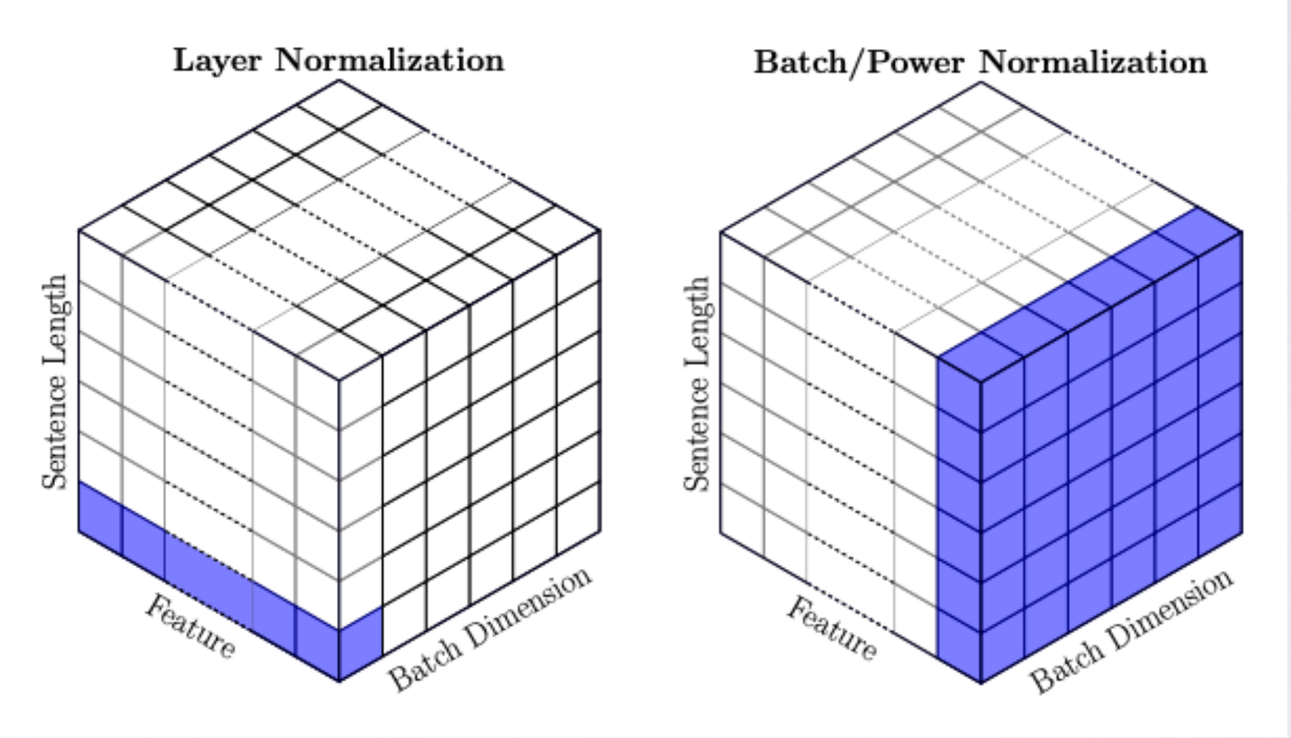
Layer normalization for a CNN:



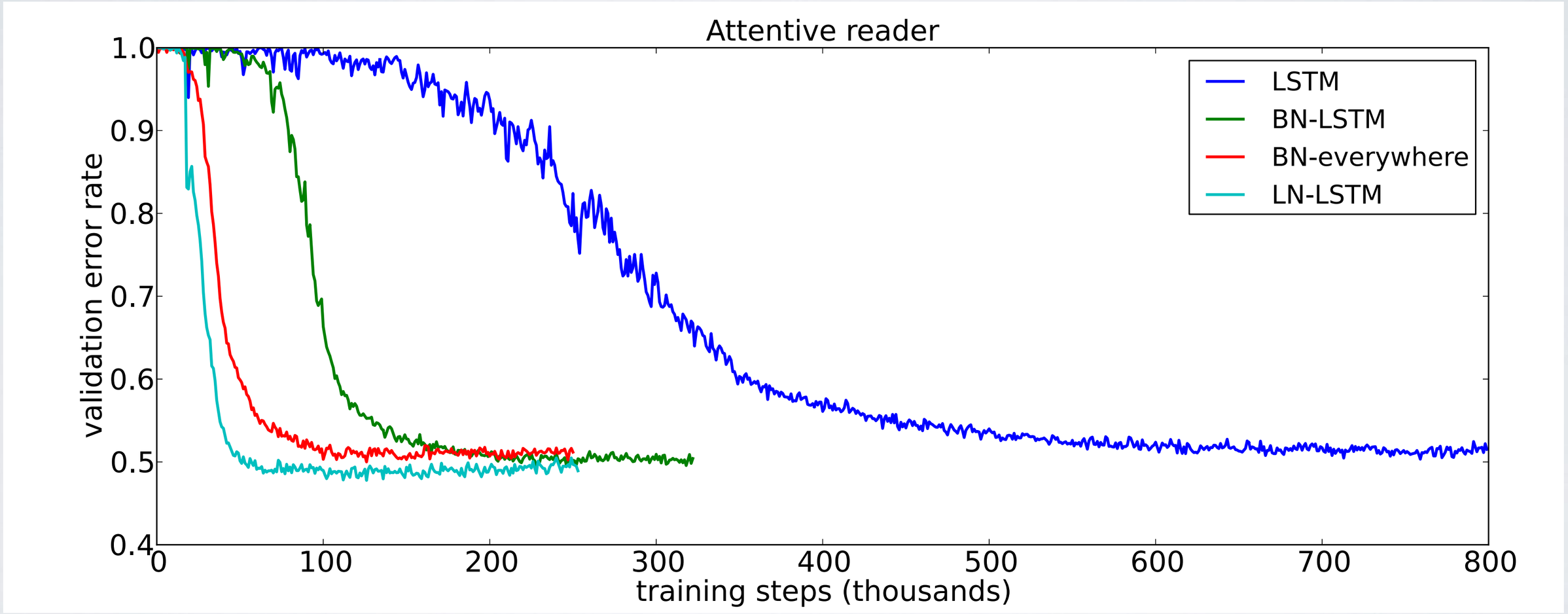
LAYER NORMALIZATION

- Widely used in sequence modeling (e.g. Transformer)

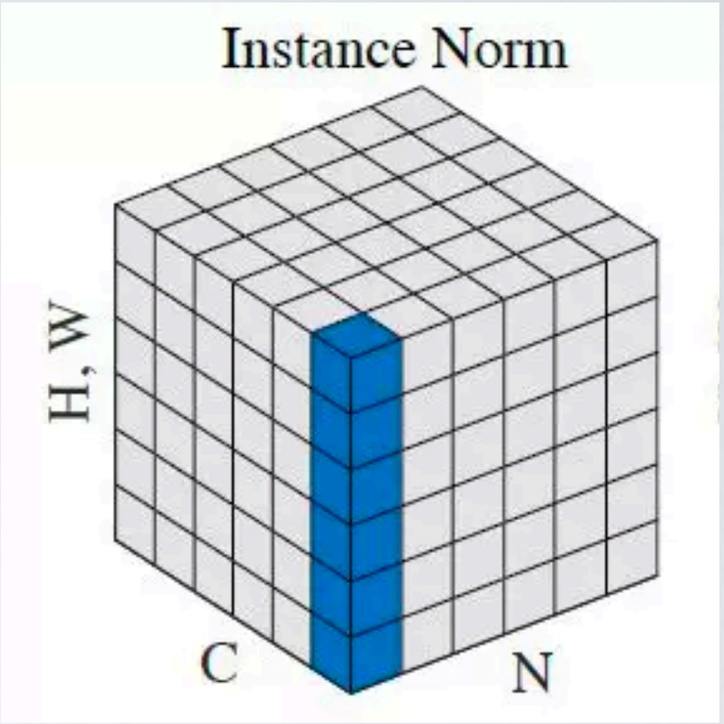
- Figure taken from Shen et al., "PowerNorm: Rethinking Batch Normalization in Transformers"



LAYER NORMALIZATION



INSTANCE NORMALIZATION



- Used for generative models such as GANs
- Especially used for generative re-stylization
  Ulyanov et al. (2017)
  Instance Normalization: The Missing Ingredient for Fast Stylization
- For Convolutional Neural Networks:

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \quad \mu_{ti} = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm}, \quad \sigma_{ti}^2 = \frac{1}{HW}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm} - mu_{ti})^2.$$

H = height
W = width
T = examples in mini-batch

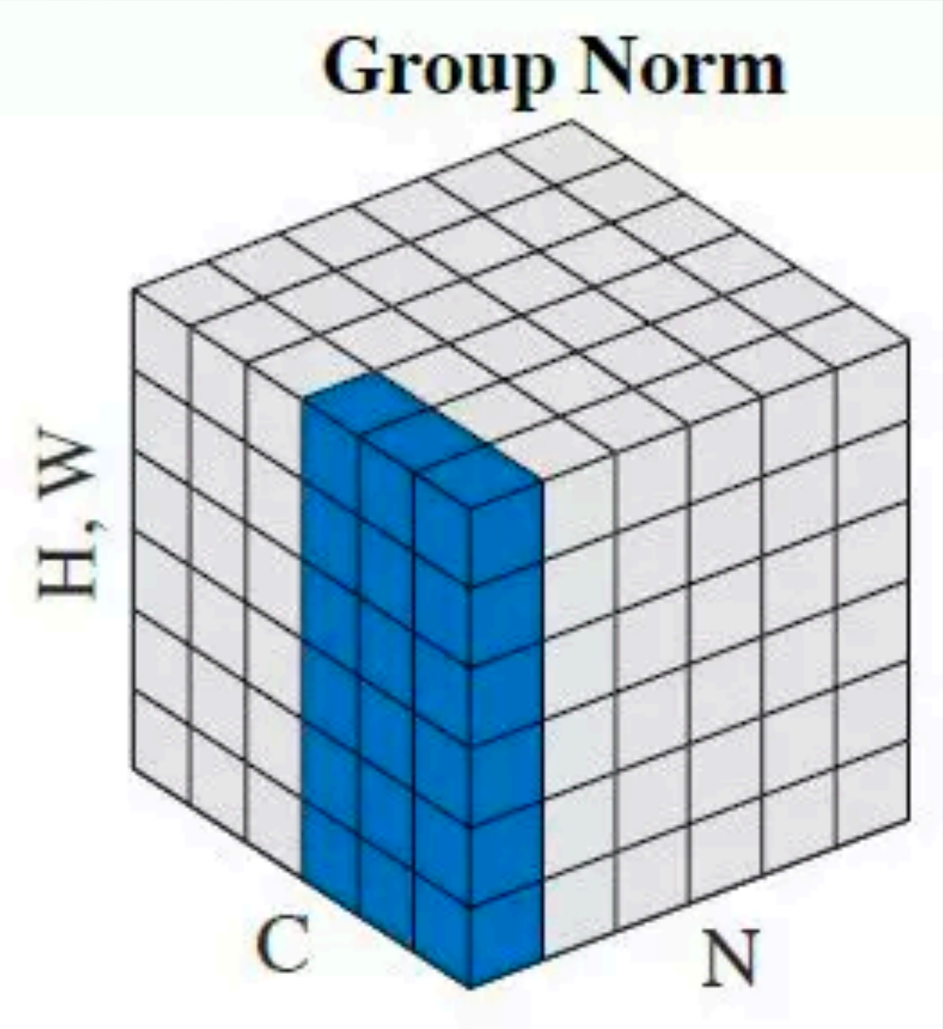
Compare with batch normalization:

$$y_{tijk} = \frac{x_{tijk} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \quad \mu_i = \frac{1}{HWT}\sum_{t=1}^{T}\sum_{l=1}^{W}\sum_{m=1}^{H} x_{tilm}, \quad \sigma_i^2 = \frac{1}{HWT}\sum_{t=1}^{T}\sum_{l=1}^{W}\sum_{m=1}^{H}(x_{tilm} - mu_i)^2.$$

GROUP NORMALIZATION

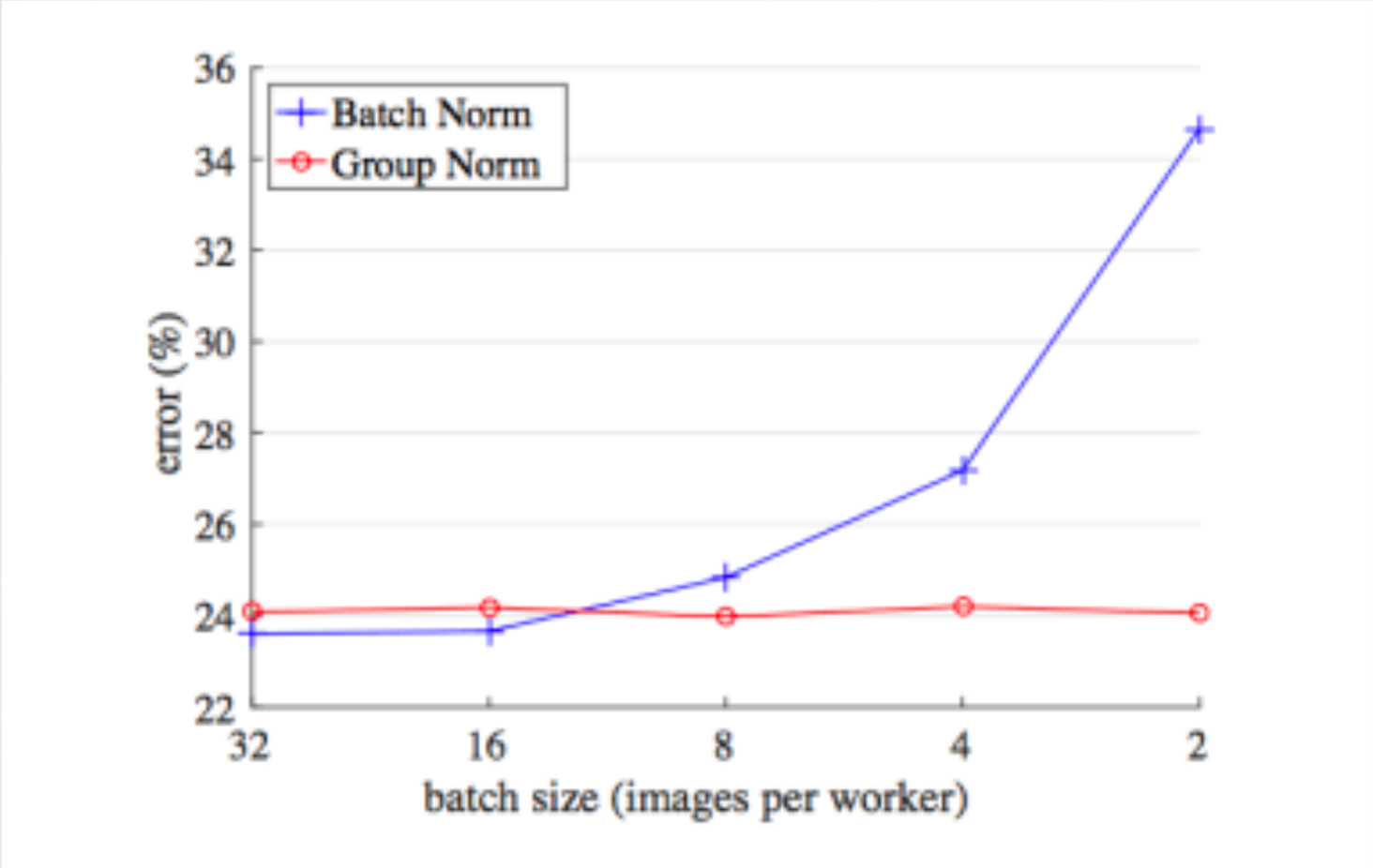
Wu and He (2018)

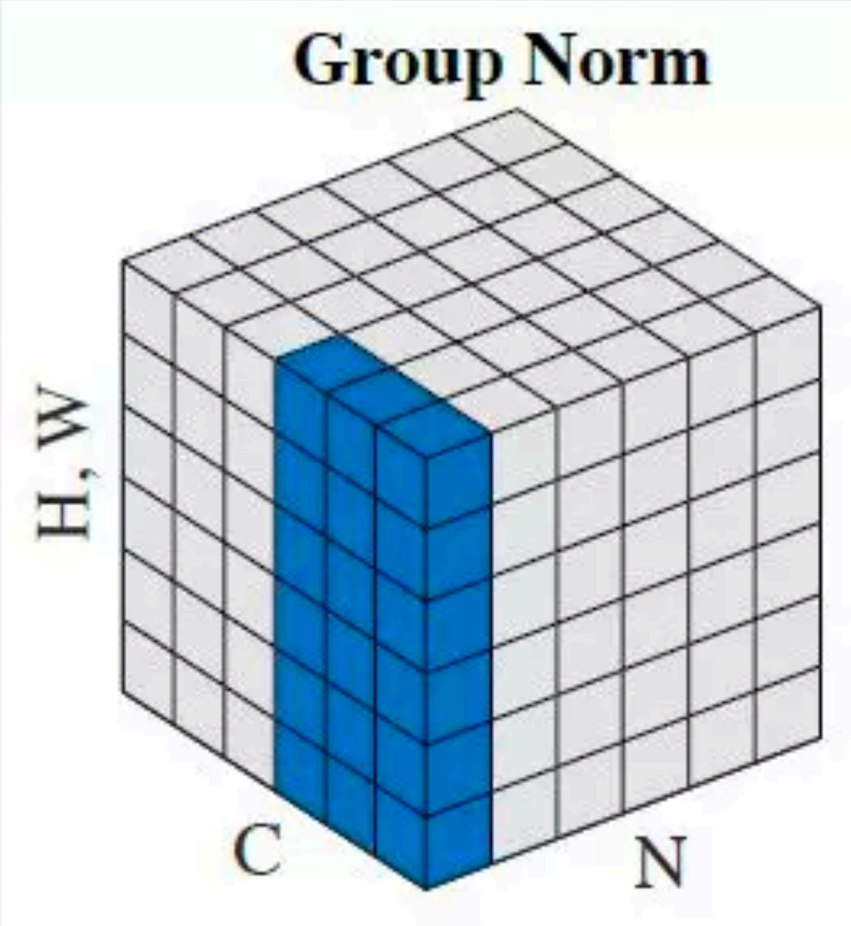
- Compromise between layer norm. And Instance norm.
- Specify groups of units or channels (in CNNs) to normalize as in layer norm.
- More flexible than layer norm. Better than batch norm when training with smaller mini-batchs



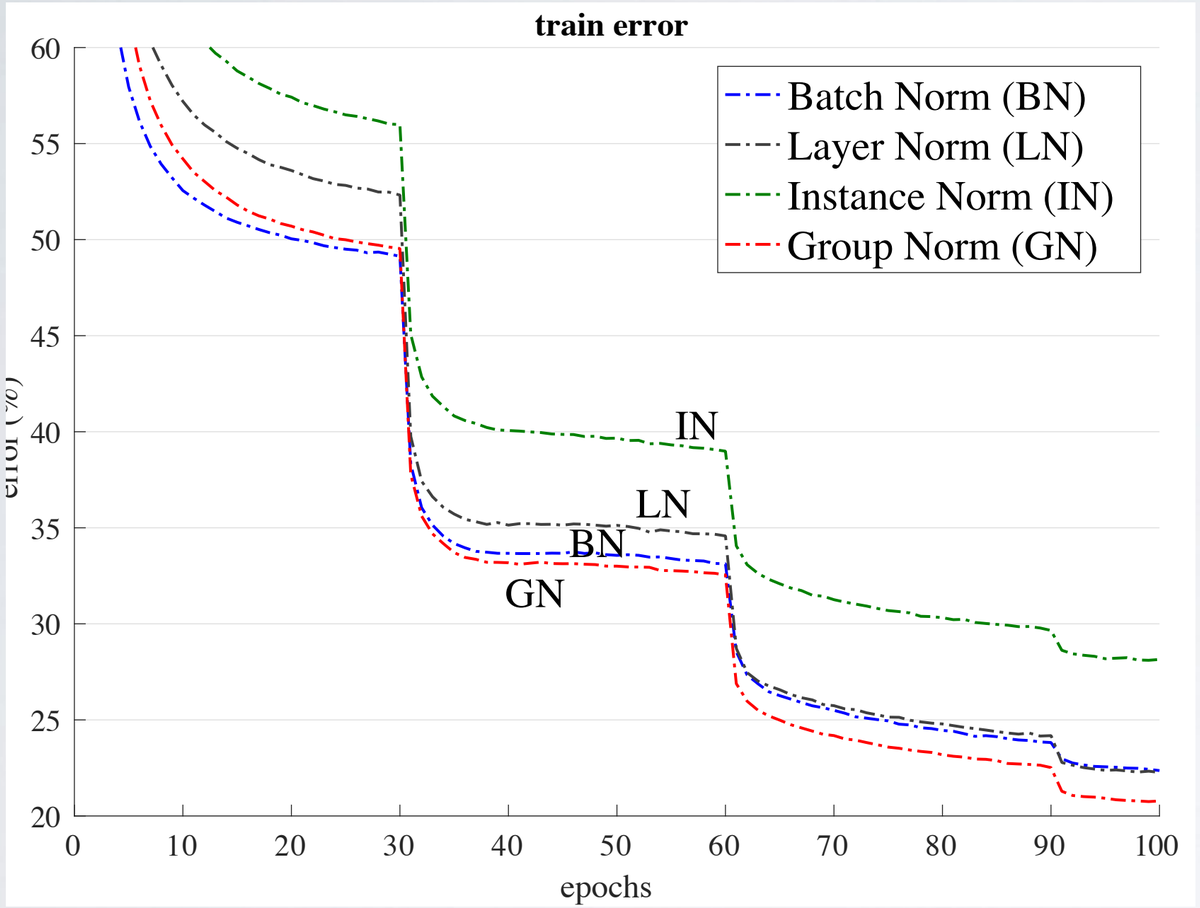
GROUP NORMALIZATION Wu and He (2018)

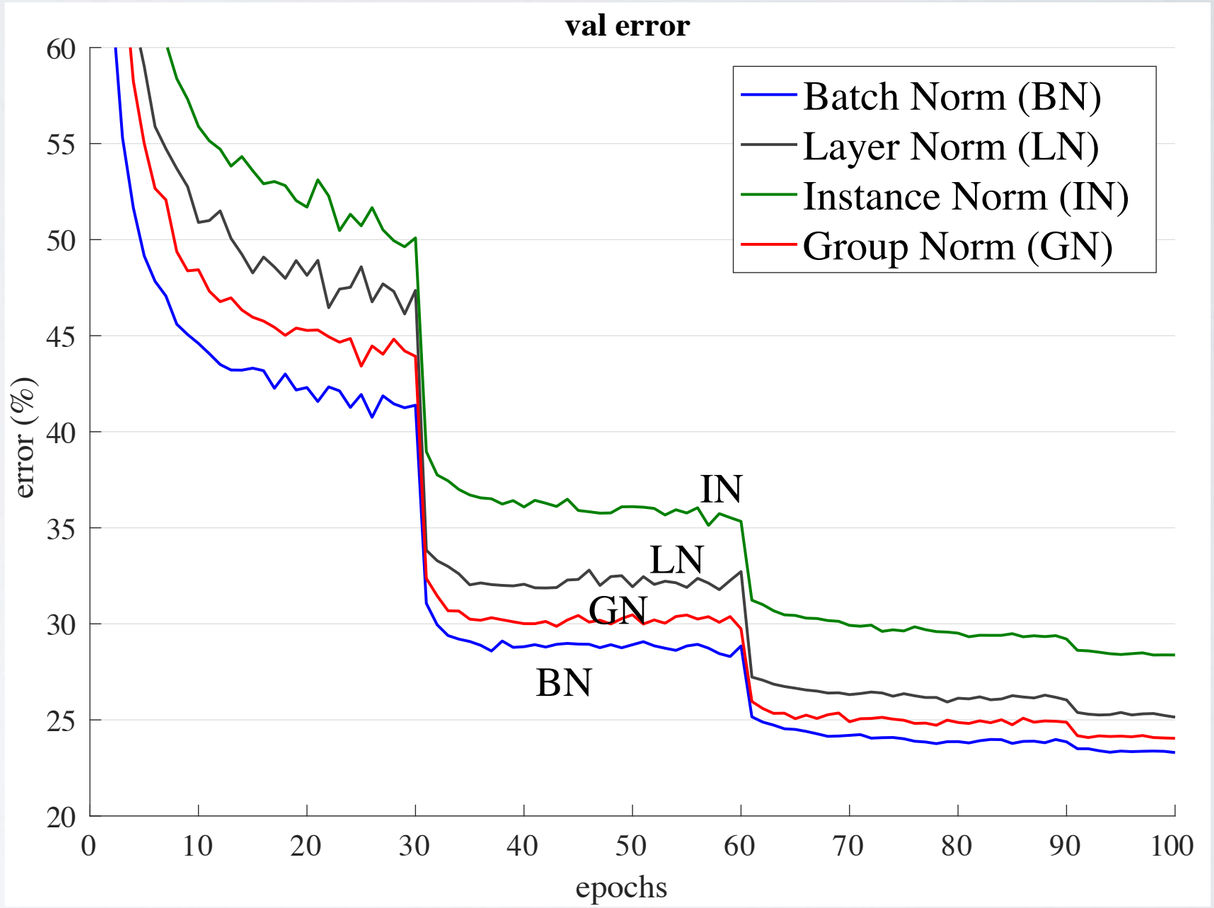
- Better than batch norm when training with smaller mini-batchs





GROUP NORMALIZATION

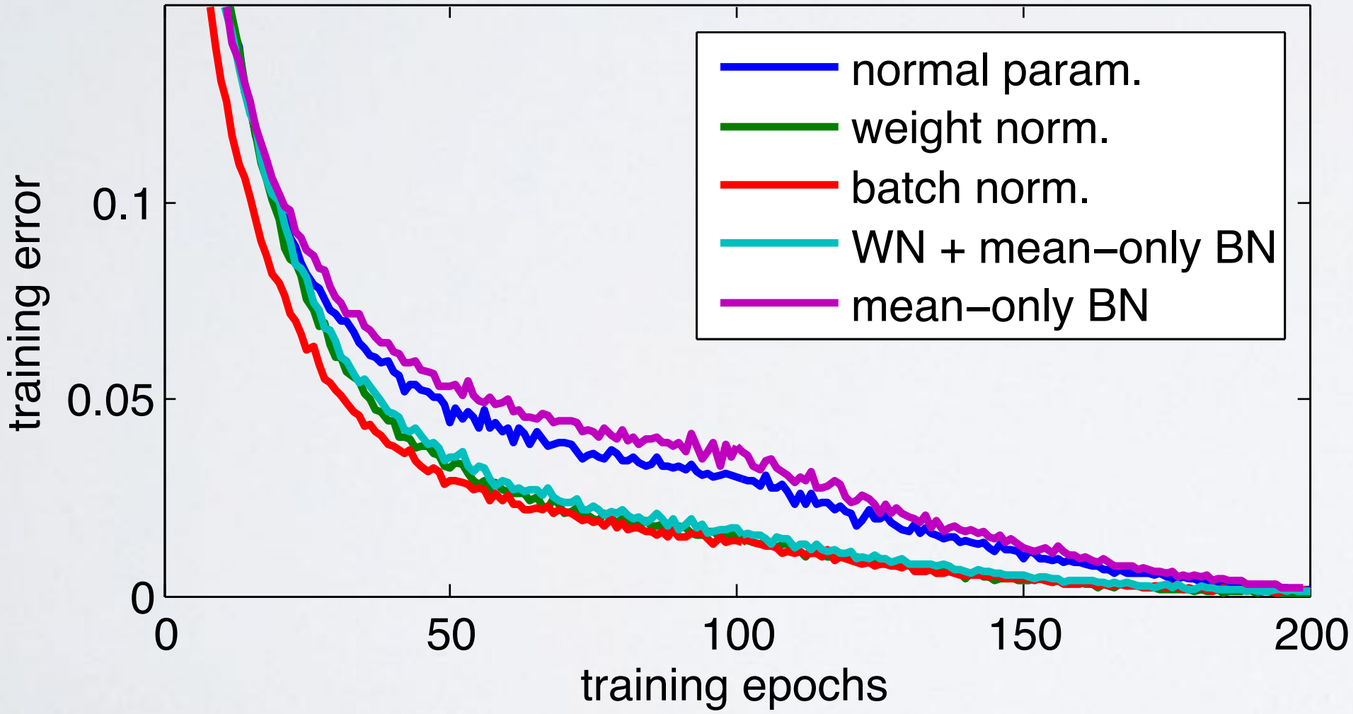




Comparison of norm. methods trained on ImageNet with batch size of 32 on ResNet50

WEIGHT NORMALIZATION

- Simple normalization method that maintains the norm of the weight vectors
- Output of a hidden unit: $y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$, where $\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$

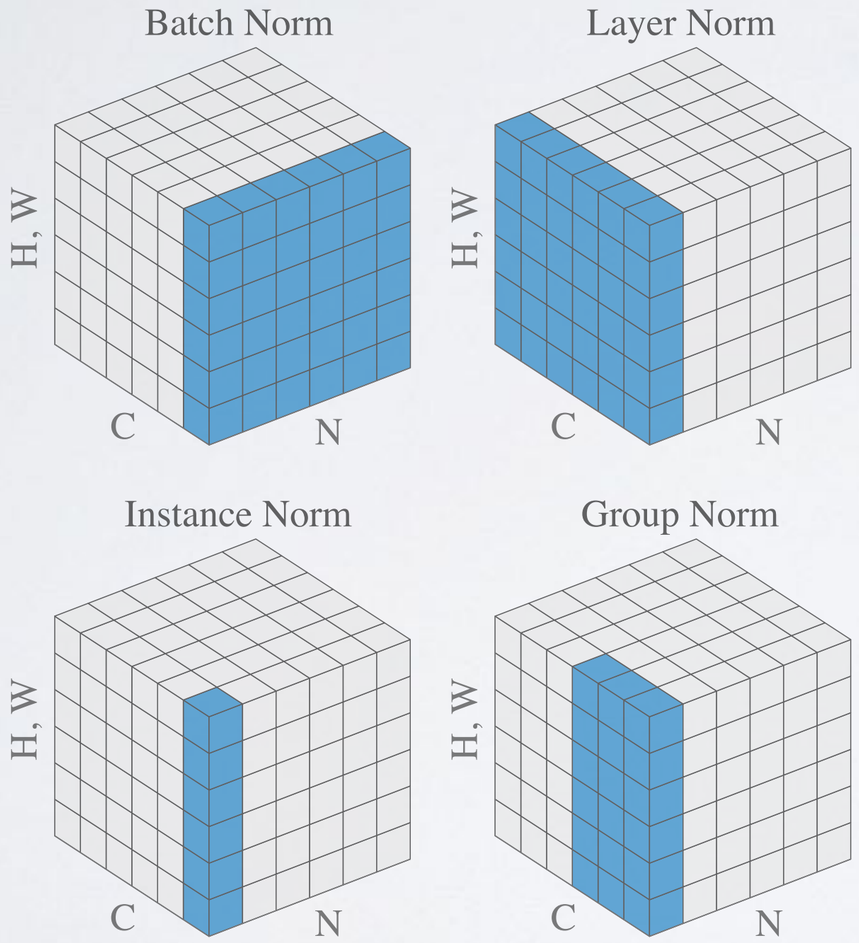


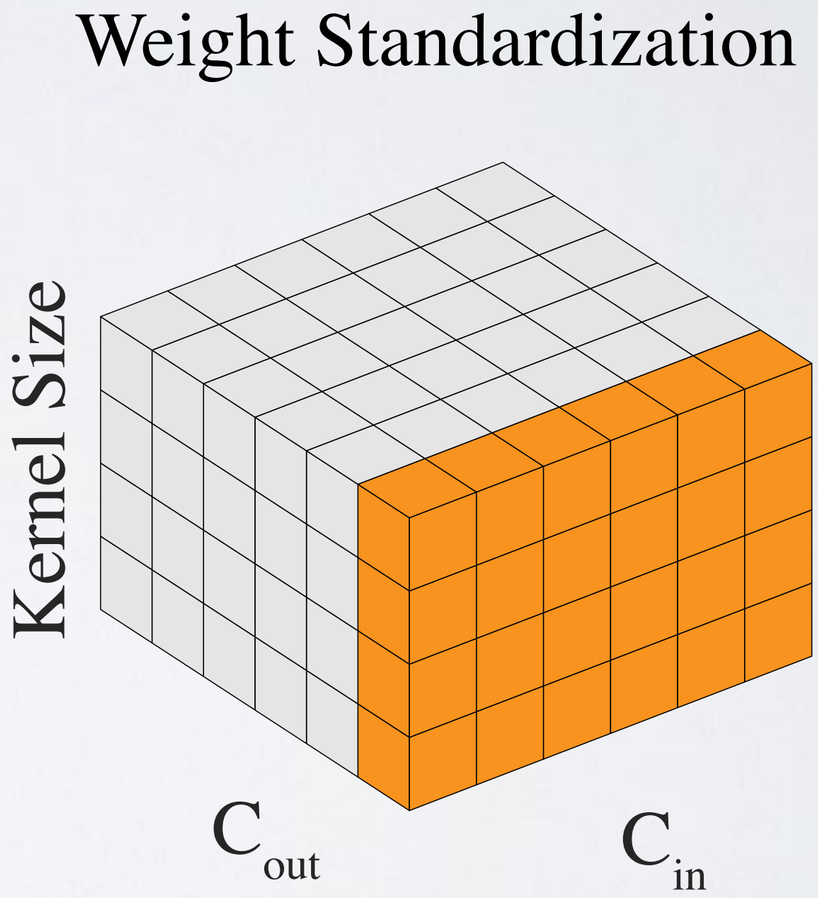
Model	Test Error
Maxout [6]	11.68%
Network in Network [17]	10.41%
Deeply Supervised [16]	9.6%
ConvPool-CNN-C [26]	9.31%
ALL-CNN-C [26]	9.08%
our CNN, mean-only B.N.	8.52%
our CNN, weight norm.	8.46%
our CNN, normal param.	8.43%
our CNN, batch norm.	8.05%
ours, W.N. + mean-only B.N.	**7.31%**

Figure 2: Classification results on CIFAR-10 without data augmentation.

WEIGHT STANDARDIZATION FOR CNNS

Qiao et al. (2019) *Micro-Batch Training with Batch-Channel Normalization and Weight Standardization*





WEIGHT STANDARDIZATION FOR CNNS

Qiao et al. (2019) *Micro-Batch Training with Batch-Channel Normalization and Weight Standardization*

$$\hat{\boldsymbol{W}} = \left[\hat{\boldsymbol{W}}_{i,j} \mid \hat{\boldsymbol{W}}_{i,j} = \frac{\boldsymbol{W}_{i,j} - \mu_{\boldsymbol{W}_{i,\cdot}}}{\sigma_{\boldsymbol{W}_{i,\cdot}}}\right],$$

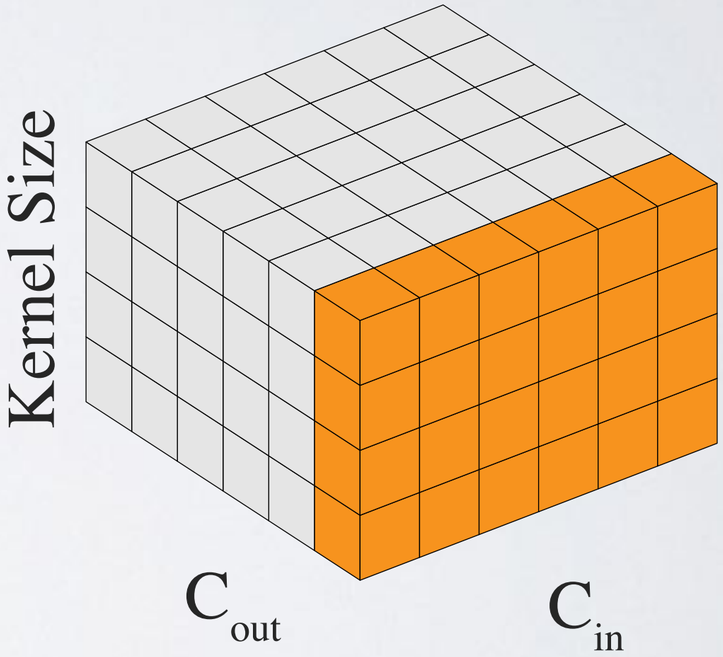
$$\boldsymbol{y} = \hat{\boldsymbol{W}} * \boldsymbol{x},$$

where $\hat{\boldsymbol{W}} \in \mathbb{R}^{O \times I}$,

$$O = C_{\text{out}} \quad \text{and} \quad I = C_{\text{in}} \times \text{Kernel\_Size}$$

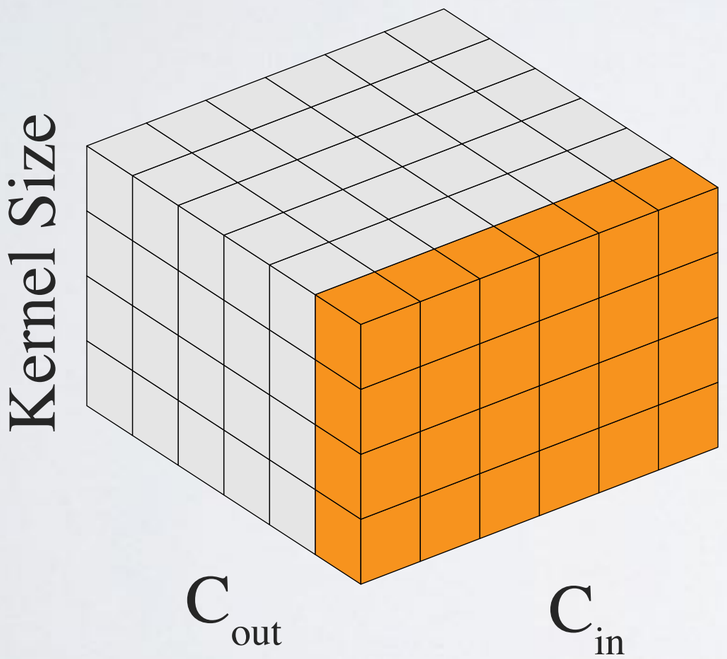
also $\mu_{\boldsymbol{W}_{i,\cdot}} = \frac{1}{I}\sum_{j=1}^{I} \boldsymbol{W}_{i,j}, \quad \sigma_{\boldsymbol{W}_{i,\cdot}} = \sqrt{\frac{1}{I}\sum_{j=1}^{I} \boldsymbol{W}_{i,j}^2 - \mu^2_{\boldsymbol{W}_{i,\cdot}} + \epsilon}.$

Weight Standardization



WEIGHT STANDARDIZATION FOR CNNS

Qiao et al. (2019) *Micro-Batch Training with Batch-Channel Normalization and Weight Standardization*



BCN (batch-channel normalization) applies BN and then GN.

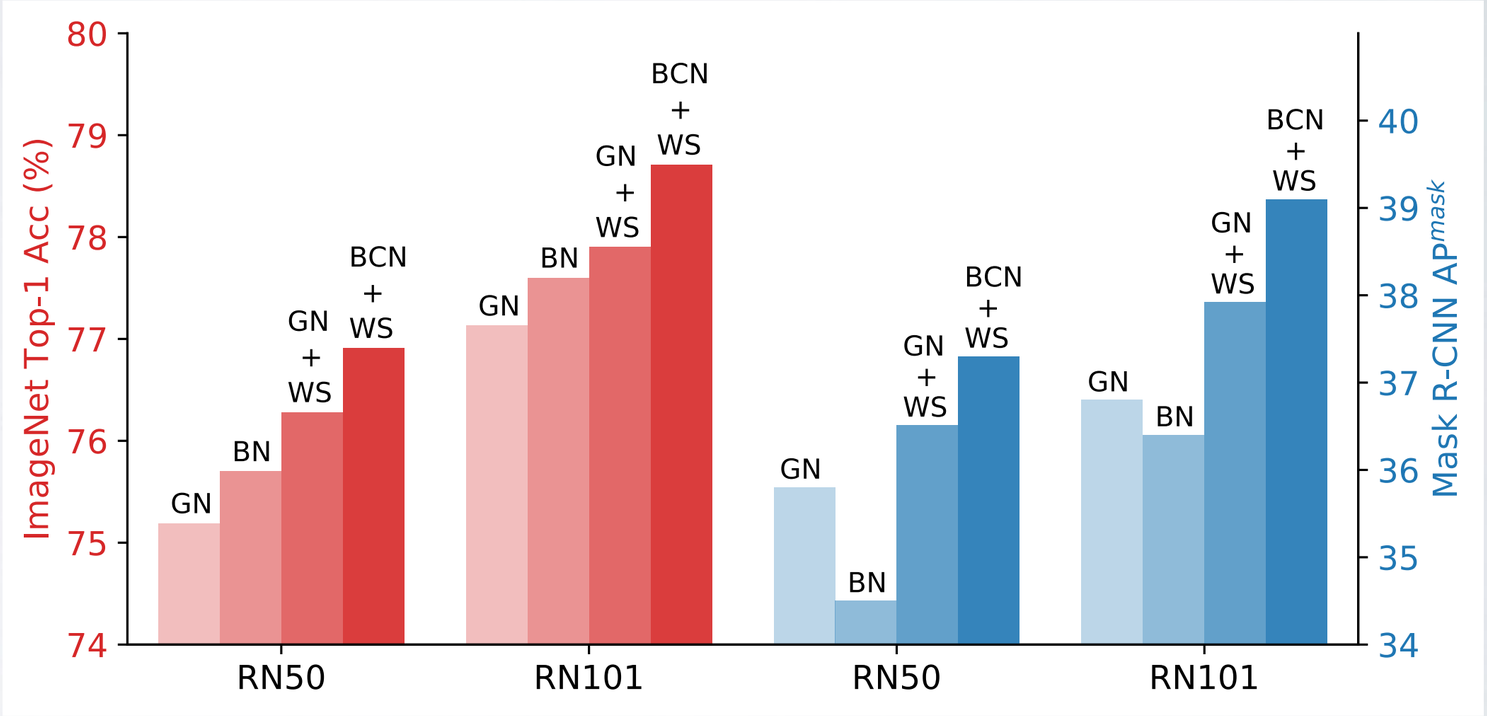


Fig. 1: Comparing BN [3], GN [6], our WS used with GN, and WS used with BCN on ImageNet and COCO. On ImageNet, BN and BCN+WS are trained with large batch sizes while GN and GN+WS are trained with 1 image/GPU. On COCO, BN is frozen for micro-batch training, and BCN uses its micro-batch implementation. GN+WS outperforms both BN and GN comfortably and BCN+WS further improves the performances.