CSC 2541: Neural Net Training Dynamics

Lecture 5 - Adaptive Gradient Methods, Normalization, and Weight Decay

Roger Grosse

University of Toronto, Winter 2021

Today

- We consider three ideas that have become staples of modern neural net training:
  1. Adaptive gradient methods (RMSprop, Adam, etc.)
  2. Normalization (esp. batch norm)
  3. Weight decay
- Deceptively simple, commonly misunderstood
- Unifying theme: you can figure out quite a lot by just reasoning about the scales of weights, activations, etc.

Batch Norm

From Ali Rahimi's classic NeurIPS 2017 Test of Time talk (emphasis mine):

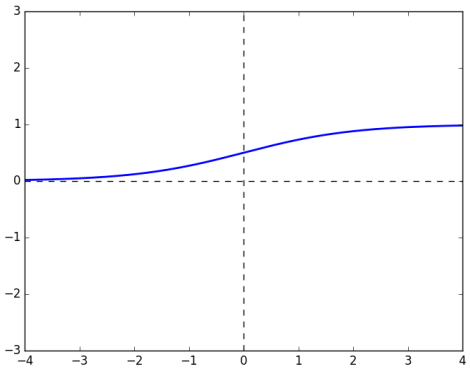
> *Batch Norm is a technique that speeds up gradient descent on deep nets. You sprinkle it between your layers and gradient descent goes faster. I think it's ok to use techniques we don't understand. I only vaguely understand how an airplane works, and I was fine taking one to this conference. But it's always better if we build systems on top of things we do understand deeply? This is what we know about why batch norm works well. But don't you want to understand why reducing internal covariate shift speeds up gradient descent? Don't you want to see evidence that Batch Norm reduces internal covariate shift? Don't you want to know what internal covariate shift is? Batch Norm has become a foundational operation for machine learning.* ***It works amazingly well. But we know almost nothing about it.***

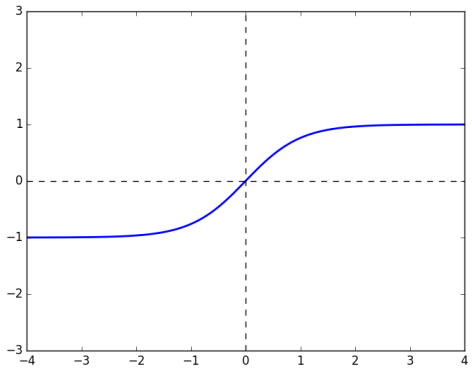
Batch Norm

- Be careful about saying "nothing" is known!
- Why should we expect "Why does batch norm help?" to have a simple answer that holds in all cases?
- Some arguments made in the original 2015 paper (and often ignored by critics):
  - Internal covariate shift leads to unstandardized activations, which hurts the conditioning
    - BN fixes this problem (by removing the ICS?)
  - Prevent dead/saturated units
  - Stochastic regularization effect caused by noisy estimates of the statistics
  - Maintain stability at high learning rates
- A more recently discovered learning rate schedule effect

Internal Covariate Shift

- **Recall from Lecture 1:** for linear regression, uncentered activations create a large outlier eigenvalue, dramatically slowing down gradient descent
- For linear regression, we can solve this by explicitly centering the features
- For neural nets, the hidden activations can become uncentered, and there's no straightforward fix
  - Pointed out by LeCun (1991)
  - The BN authors refer to this problem as Internal Covariate Shift (not a great name!)
- **Classical recommendation:** use tanh instead of logistic activations





Internal Covariate Shift

- We can make this reasoning more precise using more recent ideas
- Recall the K-FAC approximation (Lecture 4):

$$\hat{\mathbf{G}}_{\ell\ell} = \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell$$

- Uncentered activations cause an outlier eigenvalue in $\mathbf{A}_{\ell-1}$
- Recall (Lecture 4):
  - Spectral decomposition for symmetric $\mathbf{A} = \mathbf{Q_A} \mathbf{D_A} \mathbf{Q_A}^\top$ and $\mathbf{B} = \mathbf{Q_B} \mathbf{D_B} \mathbf{Q_B}^\top$

$$\mathbf{A} \otimes \mathbf{B} = (\mathbf{Q_A} \otimes \mathbf{Q_B})(\mathbf{D_A} \otimes \mathbf{D_B})(\mathbf{Q_A}^\top \otimes \mathbf{Q_B}^\top)$$

  - Therefore, if the eigenvalues of $\mathbf{A}$ are λ_i and the eigenvalues of $\mathbf{B}$ are ν_j, then the eigenvalues of $\mathbf{A} \otimes \mathbf{B}$ are the products $\lambda_i \nu_j$
  - If the corresponding eigenvectors of $\mathbf{A}$ are $\mathbf{r}_i$ and for $\mathbf{B}$ are $\mathbf{s}_j$, then the eigenvectors of $\mathbf{A} \otimes \mathbf{B}$ are $\mathbf{r}_i \otimes \mathbf{s}_j$

Internal Covariate Shift

This leads to:

ICS Conditioning Hypothesis. For a network with output dimension M, if $\mathbf{m} = \mathbb{E}[\mathbf{a}_{\ell-1}]$ is far from zero, we'd expect $\mathbf{G}_{\ell\ell}$ to have as many as M large eigenvalues, namely $\nu\lambda_i$ for each eigenvalue λ_i of $\mathbf{S}_\ell$, where $\nu = \|\mathbf{m}\|^2 + 1$. The corresponding eigenvectors are of the form $\mathbf{m} \otimes \mathbf{v}$ for some vector $\mathbf{v}$.

ICS and Invariance

- Good scientific practice: change one thing at a time
- How can we eliminate the ill-conditioning effects of ICS while changing almost nothing else?
- **Idea:** standardize the activations using an affine transformation of the parameters

$$\mathbf{a}_\ell = \phi(\mathbf{W}_\ell \mathbf{a}_{\ell-1} + \mathbf{b}_\ell)$$
$$\tilde{\mathbf{a}}_{\ell-1} = \boldsymbol{\Sigma}^{-1/2}(\mathbf{a}_{\ell-1} - \mathbf{m})$$

- Transforming the weights so that the network computes the same function:

$$\tilde{\mathbf{W}}_\ell \tilde{\mathbf{a}}_{\ell-1} + \tilde{\mathbf{b}}_\ell = \mathbf{W}_\ell \mathbf{a}_{\ell-1} + \mathbf{b}_\ell,$$

  achieved by

$$\tilde{\mathbf{W}}_\ell = \mathbf{W}_\ell \boldsymbol{\Sigma}^{1/2} \qquad \tilde{\mathbf{b}}_\ell = \mathbf{b}_\ell + \mathbf{W}_\ell \mathbf{m}$$

- Updates in this coordinate system are immune to ICS

ICS and Invariance

- Equivalent to pre-multiplying by $\mathbf{R}\mathbf{R}^\top$, where

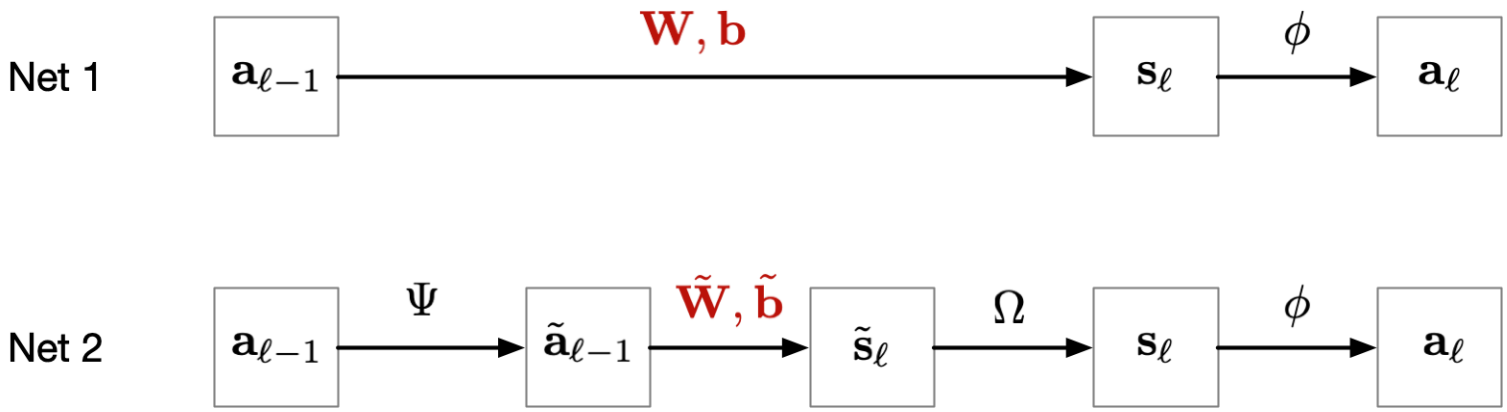
$$\mathbf{R}^{-1} = \begin{pmatrix} \boldsymbol{\Sigma}^{1/2} \otimes \mathbf{I} & \mathbf{0} \\ \mathbf{m}^\top \otimes \mathbf{I} & 1 \end{pmatrix}$$

$$[\mathbf{R}\mathbf{R}^\top]^{-1} = \begin{pmatrix} (\boldsymbol{\Sigma} + \mathbf{m}\mathbf{m}^\top) \otimes \mathbf{I} & \mathbf{m} \otimes \mathbf{I} \\ \mathbf{m}^\top \otimes \mathbf{I} & 1 \end{pmatrix}$$

- The matrix $[\mathbf{R}\mathbf{R}^\top]^{-1}$ is supposed to approximate $\mathbf{H}$.
- This matrix is "almost" diagonal, so preconditioners of this form are called quasi-diagonal
- Minimal overhead relative to ordinary neural net operations, just like diagonal preconditioning

ICS and Invariance

- Another way to arrive at quasi-diagonal preconditioners is to reason about invariance:



- If the parameters are chosen such that these two networks compute the same function, then the same should be true after running the algorithm.
  - Quasi-diagonal natural gradient is invariant to affine transformations of individual units (e.g. tanh vs. logistic)
  - K-FAC is invariant to affine transformations of the layer as a whole

Batch Normalization

Batch Normalization

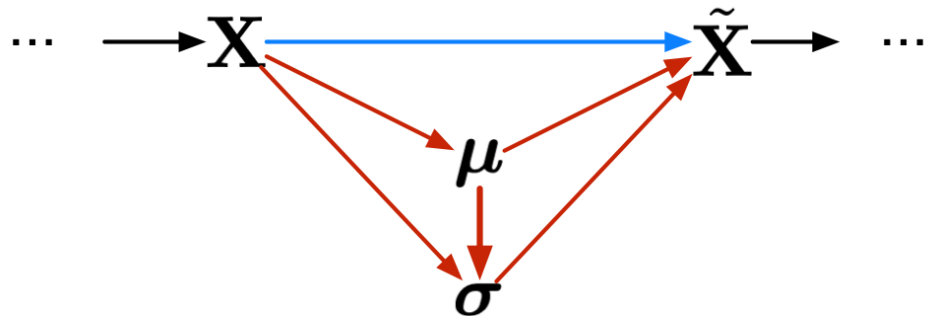
- Naïve motivation: if the architecture explicitly normalizes every unit, this eliminates ICS and improves the conditioning. Right?
- Batch normalization (BN):

$$\tilde{\mathbf{X}} = \text{BN}(\mathbf{X}) = (\mathbf{X} - \mathbf{1}\boldsymbol{\mu}(\mathbf{X})^\top) \oslash \mathbf{1}\boldsymbol{\sigma}(\mathbf{X})^\top$$

- **Training time:** Statistics are estimated from the *current* batch
- **Test time:** Use averages of training statistics
- Typically apply BN to pre-activations rather than activations
- **Note:** in practice, we fit additional parameters for the mean and variance after normalization, but I'll ignore these for this lecture

Batch Normalization

- Main difference from our preconditioning-based solution: BN is part of the architecture, so we differentiate through it
- The computation graph contains a direct path and a statistics path:



- Another difference: the statistics are estimated from the current batch, which injects noise

Batch Normalization

- Preconditioning changes the conditioning of the cost function, and nothing else
- BN also changes the effective initialization, adds stochastic regularization, completely changes the scales of the gradients, ...
- Motivations from the original paper:
  - Ameliorating the optimization effects of ICS
  - Preventing dead or saturated units
  - Maintaining stability at higher learning rates
  - Stochastic regularization
- Additionally, there's an important implicit learning rate decay effect which I believe the authors weren't aware of

A Wrinkle

A Wrinkle

- We argued that ICS creates outlier eigenvalues in the Hessian due to uncentered activations
- "ICS Conditioning Hypothesis": BN helps by removing the outlier eigenvalues
- We can also formulate:
  - ICS Removal Hypothesis: BN improves optimization by centering and/or normalizing the previous layer's activations.
- These hypotheses are logically independent
  - BN could remove the outlier eigenvalues through some means other than preventing ICS
  - Preventing ICS could have some optimization benefit other than improving conditioning

A Wrinkle

Two pieces of evidence against the ICS Removal Hypothesis:

- It generally works better to apply BN before the activation function, rather than after
- Santurkar et al. (2018) ran a sort of knockout experiment where they added ICS back in and tested if BN still improved training
  - I.e., after BN, linearly transform the activations to have some other mean and variance
  - The network still trained just as efficiently

A Wrinkle

- **Key insight:** consider what happens when BN is applied in the *next* layer, before the activation function

$$\mathbf{A}_\ell = f_\ell(\mathbf{A}_{\ell-1}, \mathbf{W}_\ell) = \phi_\ell(\mathrm{BN}(\mathbf{A}_{\ell-1}\mathbf{W}_\ell^\top))$$

- This function is invariant to rescaling and shifting $\mathbf{a}_{\ell-1}$
- If the activations are shifted, the network still computes the same function (for any particular $\mathbf{w}$), so the optimization trajectories are identical.
  - The outlier eigenvalues of have no effect!
- So the ICS Conditioning hypothesis could still be correct, even though the ICS Removal hypothesis appears to be incorrect
  - Good class project to test this

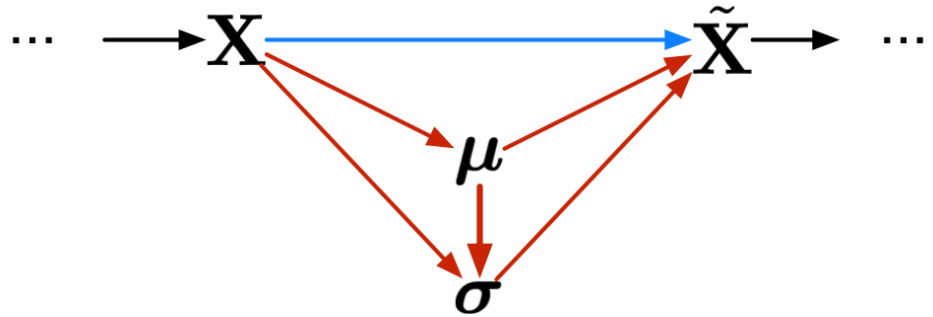
A Wrinkle

- The same result can be derived by analyzing the mechanics of backprop through the BN operation
- Consider batch centering, which centers but doesn't scale the activations
- It turns out (derivation in the readings):

$$\overline{\mathbf{W}} = \overline{\mathbf{S}}^\top \mathbf{A} \qquad \text{(without BC)}$$

$$\overline{\mathbf{W}} = \overline{\mathbf{S}}^\top \tilde{\mathbf{A}}, \qquad \text{(with BC)}$$

  where $\tilde{\mathbf{A}}$ are the centered activations
- It's the statistics path that's responsible for this effect



Implicit Learning Rate Decay

Implicit Learning Rate Decay

- Batch norm and other normalizers create an implicit learning rate decay effect
  - Even if you use a fixed learning rate, the training behaves as if you are gradually decaying the learning rate
- For this most part, this is probably beneficial — learning rate decay is very useful in stochastic optimization (Lecture 7)
- But it's a major gotcha, since you probably aren't expecting it, e.g.
  - Why does weight decay speed up optimization on the training set?
  - Different optimization algorithms can have different implicit decay schedules
  - This explains a significant fraction of confusing neural net phenomena

Implicit Learning Rate Decay

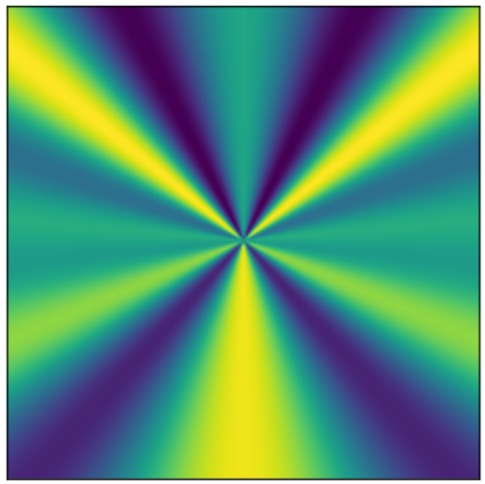
- The BN operation is invariant to rescaling the vector $\mathbf{w}_j$ of incoming weights to a unit j by a scalar $\gamma > 0$:

$$\text{BN}(\mathbf{A}\mathbf{W}^\top) = \text{BN}(\mathbf{A}\mathbf{W}^\top\boldsymbol{\Gamma}) \qquad \text{for any diagonal matrix } \boldsymbol{\Gamma} \succ \mathbf{0}.$$

- Therefore, each layer's computations, and the network as a whole, are invariant to this rescaling:

$$f_\ell(\mathbf{A}, \mathbf{W}) = f_\ell(\mathbf{A}, \boldsymbol{\Gamma}\mathbf{W})$$

- Scale invariant cost function:



Implicit Learning Rate Decay

- A function g is homogeneous of degree k if

$$g(\gamma \mathbf{x}) = \gamma^k g(\mathbf{x}) \qquad \text{for any } \mathbf{x}$$

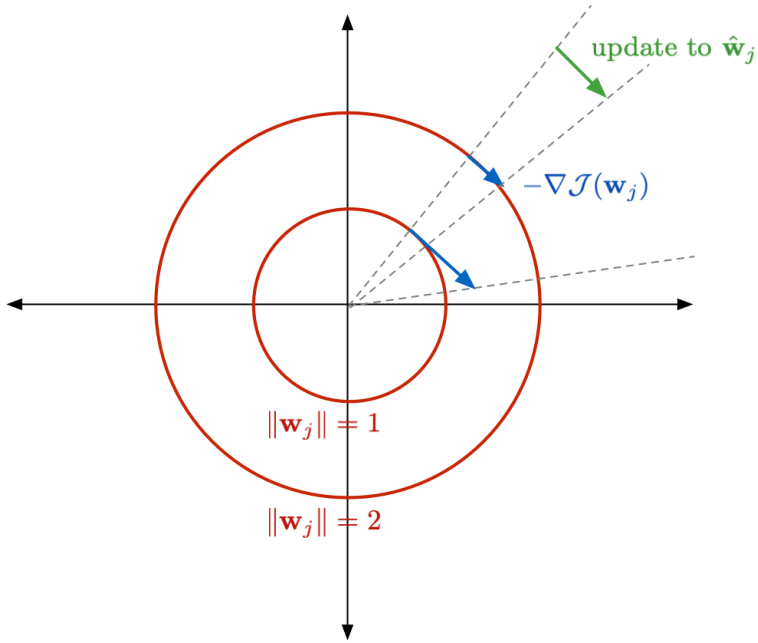
- Scale invariant = homogeneous of degree 0

- Euler showed that if g is homogeneous of degree k, then

$$\nabla g(\gamma \mathbf{x}) = \gamma^{k-1} \nabla g(\mathbf{x})$$

- Since BN is scale invariant, its gradient is homogeneous of degree -1:

$$\nabla \mathcal{J}(\gamma \mathbf{w}_j) = \gamma^{-1} \nabla \mathcal{J}(\mathbf{w}_j)$$



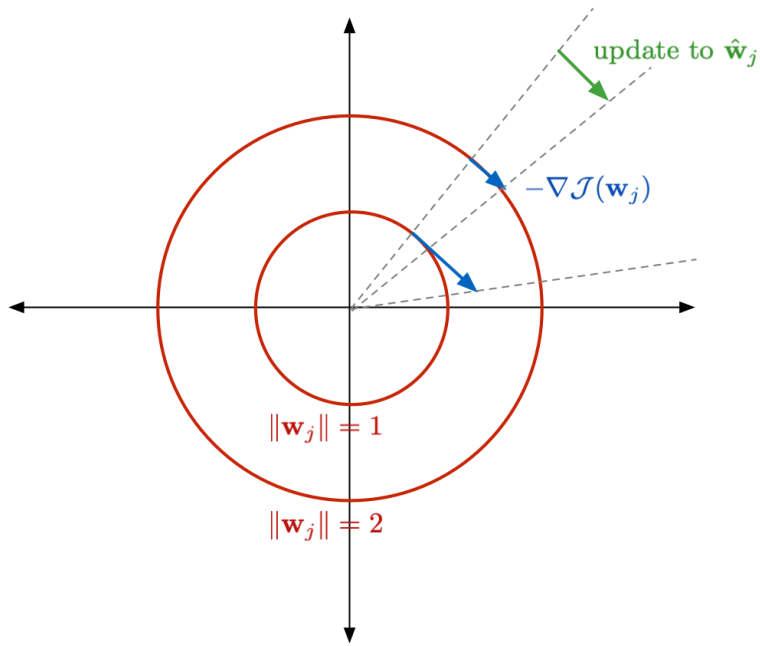
Implicit Learning Rate Decay

- Since the scale of $\mathbf{w}_j$ doesn't matter, we can canonicalize it to a unit vector:

$$\hat{\mathbf{w}}_j = \mathbf{w}_j / \|\mathbf{w}_j\|$$

- Approximating the update to $\hat{\mathbf{w}}_j$:

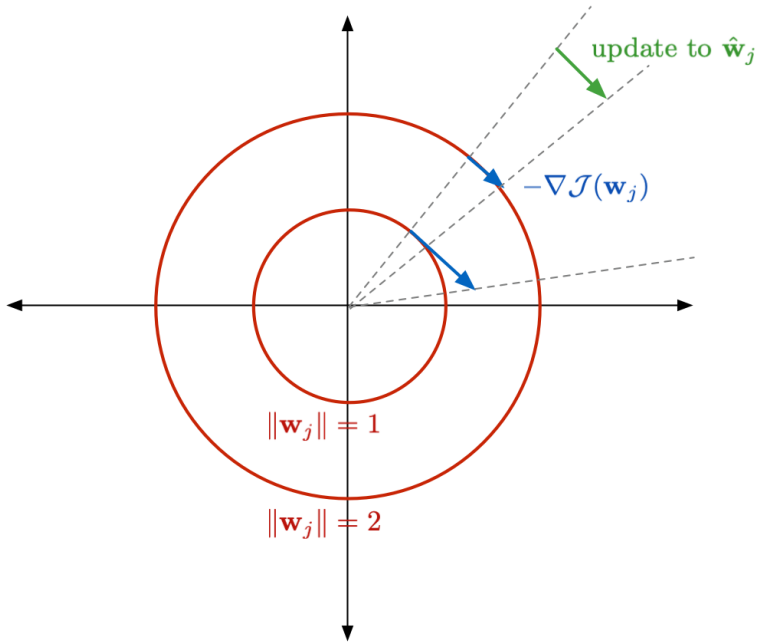
$$\begin{aligned}
\hat{\mathbf{w}}_j^{(k+1)} &= \frac{\mathbf{w}_j^{(k)} - \alpha \nabla \mathcal{J}(\mathbf{w}_j^{(k)})}{\|\mathbf{w}_j^{(k)} - \alpha \nabla \mathcal{J}(\mathbf{w}_j^{(k)})\|} \\
&\approx \frac{\mathbf{w}_j^{(k)} - \alpha \nabla \mathcal{J}(\mathbf{w}_j^{(k)})}{\|\mathbf{w}_j^{(k)}\|} \\
&= \hat{\mathbf{w}}_j^{(k)} - \alpha \|\mathbf{w}_j^{(k)}\|^{-1} \nabla \mathcal{J}(\mathbf{w}_j^{(k)}) \\
&= \hat{\mathbf{w}}_j^{(k)} - \overbrace{\alpha \|\mathbf{w}_j^{(k)}\|^{-2}}^{\text{effective LR}} \underbrace{\nabla \mathcal{J}(\hat{\mathbf{w}}_j^{(k)})}_{\text{effective gradient}}
\end{aligned}$$



Implicit Learning Rate Decay

- Observe that $\|\mathbf{w}_j\|$ increases monotonically
- Since $\mathbf{w}_j \perp \nabla \mathcal{J}(\mathbf{w}_j)$, we can apply the Pythagorean Theorem:

$$\begin{aligned}
\|\mathbf{w}_j^{(k+1)}\|^2 &= \|\mathbf{w}_j^{(k)} + \alpha \nabla \mathcal{J}(\mathbf{w}_j^{(k)})\|^2 \\
&= \|\mathbf{w}_j^{(k)}\|^2 + \alpha^2 \|\nabla \mathcal{J}(\mathbf{w}_j^{(k)})\|^2 \\
&= \|\mathbf{w}_j^{(k)}\|^2 + \frac{\alpha^2 \|\nabla \mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\|^2}{\|\mathbf{w}_j^{(k)}\|^2}
\end{aligned}$$



Implicit Learning Rate Decay

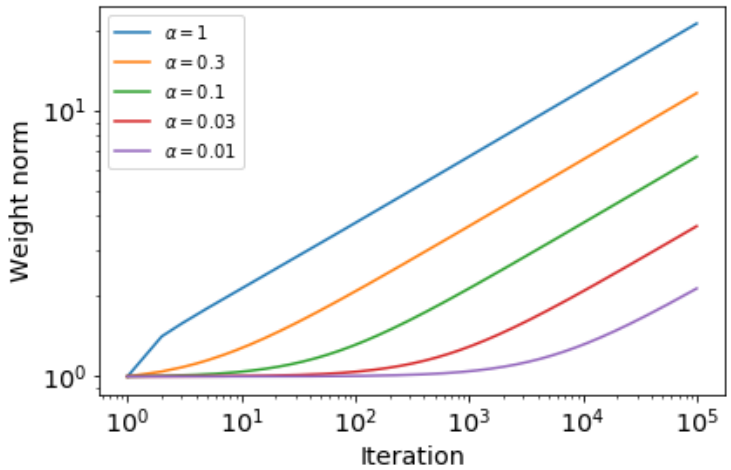
- If $\|\nabla \mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\|$ is constant throughout training (admittedly a bad assumption), then the weight norm grows roughly as:

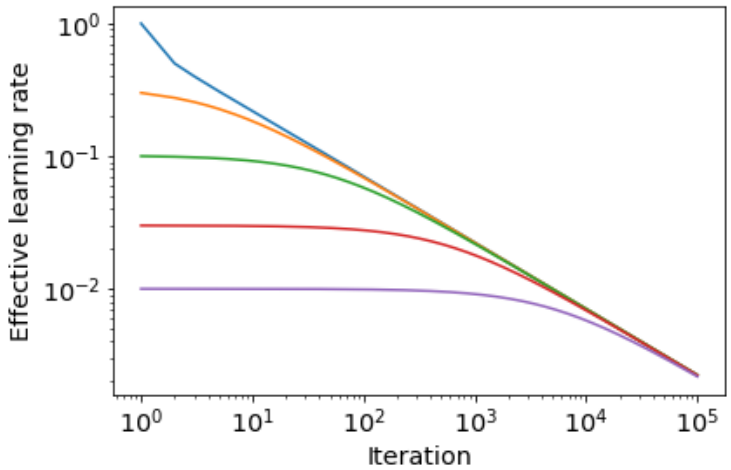
$$\|\mathbf{w}_j^{(k)}\|^2 \propto \sqrt{1 + k/k_0} \qquad \text{for some } k_0$$

- This translates into an effective learning rate schedule:

$$\hat{\alpha}_k = \frac{\hat{\alpha}_0}{\sqrt{1 + k/k_0}}$$

- The explicit learning rate hyperparameter α gives you surprisingly little control over the effective learning rate





Implicit Learning Rate Decay

- Bengio (2012):

  The [learning rate] is often the single most important hyperparameter and one should always make sure that it has been tuned (up to approximately a factor of 2)... If there is only time to optimize one hyper-parameter and one uses stochastic gradient descent, then this is the hyper-parameter that is worth tuning.

- Today: not a big deal
- **Aside:** Cohen et al., "Gradient descent in neural networks typically occurs at the edge of stability" showed that even in non-BN networks, for full-batch gradient descent, the network's curvature adapts to compensate for the learning rate, through a completely different mechanism!

Implicit Learning Rate Decay

Counteracting the implicit LR effect:

- Exponentially increasing LR schedule (Li and Arora, 2020)
- Use weight decay (up next!)
- Explicitly normalize each $\mathbf{w}_j$ to unit norm
  - Then the effective LR is just α
  - This is not common practice, but I expect it could eliminate a lot of experimental confounds

Implicit Learning Rate Decay

- The above argument suggests that BN implements a $\mathcal{O}(1/\sqrt{k})$ decay schedule
- Some reasons this is not exactly true:
  - Above analysis assumes $\|\nabla \mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\|$ is constant, which is hopefully not the case (if you succeeded in learning anything!)
  - Separate learning rate for each unit (so features that have already changed a lot get slowed down more) — maybe a sort of feedback control
  - Effect doesn't apply to the output layer, which doesn't feed into BN — therefore, the output layer trains faster later in training
  - Different algorithms can have different decay schedules
    - E.g. K-FAC is invariant to affine transformations, and therefore immune to this effect

More coming...