Chapter 5
Adaptive Gradient Methods, Normalization, and Weight Decay

Roger Grosse

1 Introduction

This lecture considers three staples of modern deep learning systems: adaptive gradient methods (such as RMSprop and Adam), normalization layers (such as batch norm, weight norm, and layer norm), and weight decay. While the three parts of the lecture are somewhat independent, there are some threads that tie them together. All three components create surprising interactions between all the various knobs we have to tune to get a deep learning system to work. And, for the most part, one can reason pretty well about the behavior of these components using only elementary arithmetic; no fancy math required.

The second component, normalization layers, is a particularly interesting case to consider because it's often served as the canonical example of a deep learning architecture we don't understand. To begin with, consider Rahimi (2017)'s NeurIPS 2017 Test of Time talk, which was remarkably well-delivered and entertaining, and launched a conversation about the state of scientific rigor in our field. He characterized the state of deep learning research as "alchemy," in the sense of having discovered a wide array of pratical and useful tricks, but lacking a foundational understanding of the principles (analogously to not having an atomic theory). He cited BN, and in particular its justification in terms of ICS, as a specific example of this (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.**

Clearly, this was a somewhat lighthearted talk that didn't put heavy emphasis on nuance. The claim that we know "almost nothing" about BN was probably not meant to be taken literally. However, it's been taken both seriously and literally by much of our community as a statement about our then-current (i.e. as of 2017) best scientific understanding. Perhaps as a result, a lot of researchers don't appreciate what was already understood about normalization even at the time.

There is indeed a sense in which we don't understand BN, but not the one which is widely believed. Contra the "alchemy" claim, the problem isn't that we lack an atomic theory. The problem, rather, is that neural nets have many moving parts, which makes it challenging to reason through the consequences of the theory in any particular situation. Asking "why does BN help?" is like asking an organic chemist, "what does nitrogen do?" The chemist could discuss nitrogen's role in particular molecules and reactions, perhaps down to the quantum level if needed, and abstract away some general patterns and principles. In cases where its role isn't understood, they could suggest experiments that could disentangle different hypotheses. But it is impossible to formulate a succinct list of rules to explain nitrogen's role in all the situations one is likely to encounter. Similarly, the original BN paper (Ioffe and Szegedy, 2015) listed a variety of effects BN might have, all of which could be analyzed in more detail using principles which were understood at the time. I believe their analysis was very close to the modern understanding. But it is hard to give a succinct summary of what BN does, because there are multiple effects going on, and which ones are important in any particular case will depend on the specific task, optimization algorithm, and architecture.

The goal of this lecture is to give you the tools you need to reason about adaptive gradient methods, normalization, and weight decay. Most of the analysis will be fairly elementary, only involving basic arithmetic. Most of it will appear obvious in hindsight. However, it's a very important lecture, because it contains a lot of gotchas that arise when trying to understand a deep learning system. I said earlier in this course that half of neural net phenomena can be explained by reasoning about linear networks. I'd estimate that half of the *remaining* phenomena can be explained using the ideas covered in this lecture. It's likely that, after internalizing the ideas in this lecture, you'll start to notice a lot of unaccounted-for confounds in papers purporting to explain neural net phenomena.

2 Adaptive Gradient Methods

The main workhorse of neural net training is gradient-based optimization, where the gradients are computed using backpropagation. A significant fraction of the time, the optimization is done using stochastic gradient descent (SGD), or SGD with momentum (covered in Chapter 9). Most of the remainder is made up of **adaptive gradient methods**, which rescale each individual entry of the gradient according to a running average of its magnitude. Prominent examples include RMSprop (Tieleman and Hinton, 2012), Adagrad (Duchi et al., 2011), and Adam (Kingma and Ba, 2015).

Let's start with **RMSprop** (Tieleman and Hinton, 2012), which applies

the following update rule. Here, $\oslash$ denotes elementwise division, and squares and powers are taken elementwise.

Iterations are denoted with subscripts k rather than superscripts to avoid clutter.

$$\begin{aligned}\mathbf{g}_k &\leftarrow \nabla \mathcal{J}_k(\mathbf{w}_k) \\ \mathbf{s}_k &\leftarrow \beta \mathbf{s}_{k-1} + (1-\beta)\mathbf{g}_k^2 \\ \mathbf{w}_k &\leftarrow \mathbf{w}_{k-1} - \alpha \mathbf{g}_k \oslash \sqrt{\mathbf{s}_k + \epsilon \mathbf{1}}\end{aligned}$$

The interpretation is simple: $\mathbf{s}$ is an exponential moving average of the squared gradient $\mathbf{g}_k^2$. The final step is like the SGD update, except that the update is scaled inversely proportional to $\sqrt{\mathbf{s}_k + \epsilon \mathbf{1}}$, the estimate of the standard deviation of the gradient. (The $\epsilon \mathbf{1}$ in the denominator is a small positive number added for stability.)

The hyperparameter β can be seen as determining the timescale. The squared gradients are averaged over approximately the last $(1-\beta)^{-1}$ iterations.

One way to understand RMSprop is that the updates to each weight are scaled such that their magnitude is approximately α. Why is this useful? For ordinary SGD, individual derivatives might be very large or very small, and taking too large or too small an update can be problematic for optimization. By ensuring that each update has magnitude approximately α (for a small value of α such as 10^{-3}), we ensure that each weight is changed by only a little bit in each iteration, but over many (e.g. 1000) iterations, the weights still have the opportunity to move a long distance.

The entrywise scale factors are sometimes referred to as **adaptive learning rates**, but we need to be careful with this interpretation. RMSprop still has a learning rate parameter α, and performance is still sensitive to the choice of α, albeit less so than for SGD.

RMSprop is closely related to **Adagrad** (Duchi et al., 2011), an algorithm for online convex optimization. The update for Adagrad is broadly similar to that of RMSprop:

$$\begin{aligned}\mathbf{g}_k &\leftarrow \nabla \mathcal{J}_k(\mathbf{w}_k) \\ \mathbf{f}_k &\leftarrow \mathbf{f}_{k-1} + \mathbf{g}_k^2 \\ \mathbf{w}_k &\leftarrow \mathbf{w}_{k-1} - \alpha \mathbf{g}_k \oslash \sqrt{\mathbf{f}_k + \epsilon \mathbf{1}}\end{aligned}$$

The main difference from RMSprop is that Adagrad maintains the sum of squared gradients $\mathbf{f}_k = \sum_{\kappa=1}^{k} \mathbf{g}_\kappa^2$, rather than the exponential moving average $\mathbf{s}_k$. Observe that the sum and the average have different scales: if the gradient scale remains consistent, then they will differ by approximately a factor of k. Hence, the Adagrad update to the weights will differ from the RMSprop update by a scale factor of $\mathcal{O}(1/\sqrt{k})$. This means Adagrad implements an implicit learning rate decay schedule, which is essential for proving convergence in the online convex optimization setting, but which in practice can be either helpful or harmful for neural net optimization, depending on the task and architecture. One reason RMSprop and Adam are generally preferred to Adagrad is likely the lack of an implicit learning rate decay schedule.

We'll see in Section 4.3 that normalization layers cause an implicit learning rate decay, also of $\mathcal{O}(1/\sqrt{k})$ in the case of SGD. This could be why the Adagrad decay schedule, which is so important for convergence in the online convex optimization setting, doesn't seem to help for neural net optimization.

Adam (Kingma and Ba, 2015) is a variant of RMSprop which includes a sort of heavy ball momentum, and also uses a slightly different estimator

of the variances which avoids a certain bias at the start of training.

$$\begin{aligned}
\mathbf{g}_k &\leftarrow \nabla \mathcal{J}_k(\mathbf{w}_{k-1}) \\
\mathbf{m}_k &\leftarrow \beta_1 \mathbf{m}_{k-1} + (1-\beta_1)\mathbf{g}_k \\
\mathbf{s}_k &\leftarrow \beta_2 \mathbf{s}_{k-1} + (1-\beta_2)\mathbf{g}_k^2 \\
\hat{\mathbf{m}}_k &\leftarrow \mathbf{m}_k / (1-\beta_1^k) \\
\hat{\mathbf{s}}_k &\leftarrow \mathbf{s}_k / (1-\beta_2^k) \\
\mathbf{w}_k &\leftarrow \mathbf{w}_{k-1} - \alpha \hat{\mathbf{m}}_k \oslash (\sqrt{\hat{\mathbf{s}}_k} + \epsilon \mathbf{I})
\end{aligned}$$

Note that in the denominator for the weight update, ϵ is outside the square root for Adam, while it is inside the square root for RMSprop and Adagrad. As far as I know, this doesn't reflect any interesting difference between the algorithms. But it is a minor gotcha when doing algorithmic comparisons, since the hyperparameter ϵ has a different scale between the algorithms.

Observe that $\mathbf{m}_k$ is an exponential moving average of the gradients. Using $\mathbf{m}_k$ for the weight update instead of $\mathbf{g}_k$ is essentially like heavy ball momentum (discussed in Chapter 9). The hyperparameter β_1 functions like the momentum decay parameter, hence the default value is 0.9. The hyperparameter β_2 is like β in RMSprop, and the default value is 0.999 (correspoding to a timescale of 1000 steps for averaging the gradient magnitudes). The bias-corrected moments $\hat{\mathbf{m}}_k$ and $\hat{\mathbf{s}}_k$ are intended to compensate for the fact that the moments are initialized to zero (and hence the estimates would otherwise be too small at the start of training).

One gotcha is that α has a different interpretation from that of ordinary heavy ball momentum. In heavy ball momentum, the effective learning rate (terminal velociy) is $\alpha(1-\beta)^{-1}$, while in Adam, the effective learning rate is α. See Chapter 9.

2.1 Use of Second-Order Information

Adaptive gradient methods are sometimes viewed as approximate second-order optimizers. To understand why, recall our discussion of the true and empirical Fisher information matrices from Chapter 3. The true Fisher information matrix

$$\mathbf{F} = \mathbb{E}_{\substack{\mathbf{x}\sim p_{\text{data}} \\ \mathbf{t}\sim r(\cdot\,|\,\mathbf{x})}}[\mathcal{D}\mathbf{w}\mathcal{D}\mathbf{w}^\top] \tag{1}$$

is the covariance of the log-likelihood gradients when the targets are sampled *from the model's output distribution*. We saw that $\mathbf{F}$ is equivalent to the Gauss-Newton Hessian $\mathbf{G}$ when the output layer represents the natural parameters of an exponential family, and the loss is its negative log-likelihood. In Chapter 2, we saw that under certain conditions, $\mathbf{G}$ is a good approximation to the true Hessian $\mathbf{H}$. Hence, $\mathbf{F}$ can be treated as an approximation to $\mathbf{H}$.

In Chapter 3, we also briefly introduced the empirical Fisher matrix. In the case where the batch size is 1, the matrix is given by:

$$\begin{aligned}
\mathbf{F}_{\text{emp}} &= \mathbb{E}_{(\mathbf{x},\mathbf{t})\sim p_{\text{data}}}[\nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w}) \nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})^\top] \\
&= \text{Cov}_{(\mathbf{x},\mathbf{t})\sim p_{\text{data}}}(\nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})) + \nabla \mathcal{J}(\mathbf{w}) \nabla \mathcal{J}(\mathbf{w})^\top,
\end{aligned} \tag{2}$$

Note that, unlike with the true Fisher matrix, the $\nabla \mathcal{J}(\mathbf{w}) \nabla \mathcal{J}(\mathbf{w})^\top$ term does not drop out because the cost gradient may be nonzero.

where $\mathcal{J}(\mathbf{w}) = \mathbb{E}_{(\mathbf{x},\mathbf{t})\sim p_{\text{data}}}[\mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})]$ is the full batch cost. Like with the true Fisher, this is an uncentered covariance matrix of gradients. The difference is that it uses the training labels, rather than samples from the output distribution. The second moments estimated by RMSprop and Adam can be seen as estimates of the diagonal entries of $\mathbf{F}_{\text{emp}}$. (A summary of all the matrices covered in this course is given in Figure 1.)

The second term, $\nabla \mathcal{J}(\mathbf{w}) \nabla \mathcal{J}(\mathbf{w})^\top$, is rank-1. But because algorithms like RMSprop and Adam use an exponential moving average of the second-order statistics, in practice its contribution to $\mathbf{F}_{\text{emp}}$ will have rank larger than 1.

Unfortunately, the argument that $\mathbf{F} \approx \mathbf{H}$ doesn't carry over to $\mathbf{F}_{\text{emp}}$. The two matrices are expected to be similar if the model is close to the optimum (so that $\nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})$ is approximately zero-mean), there is significant

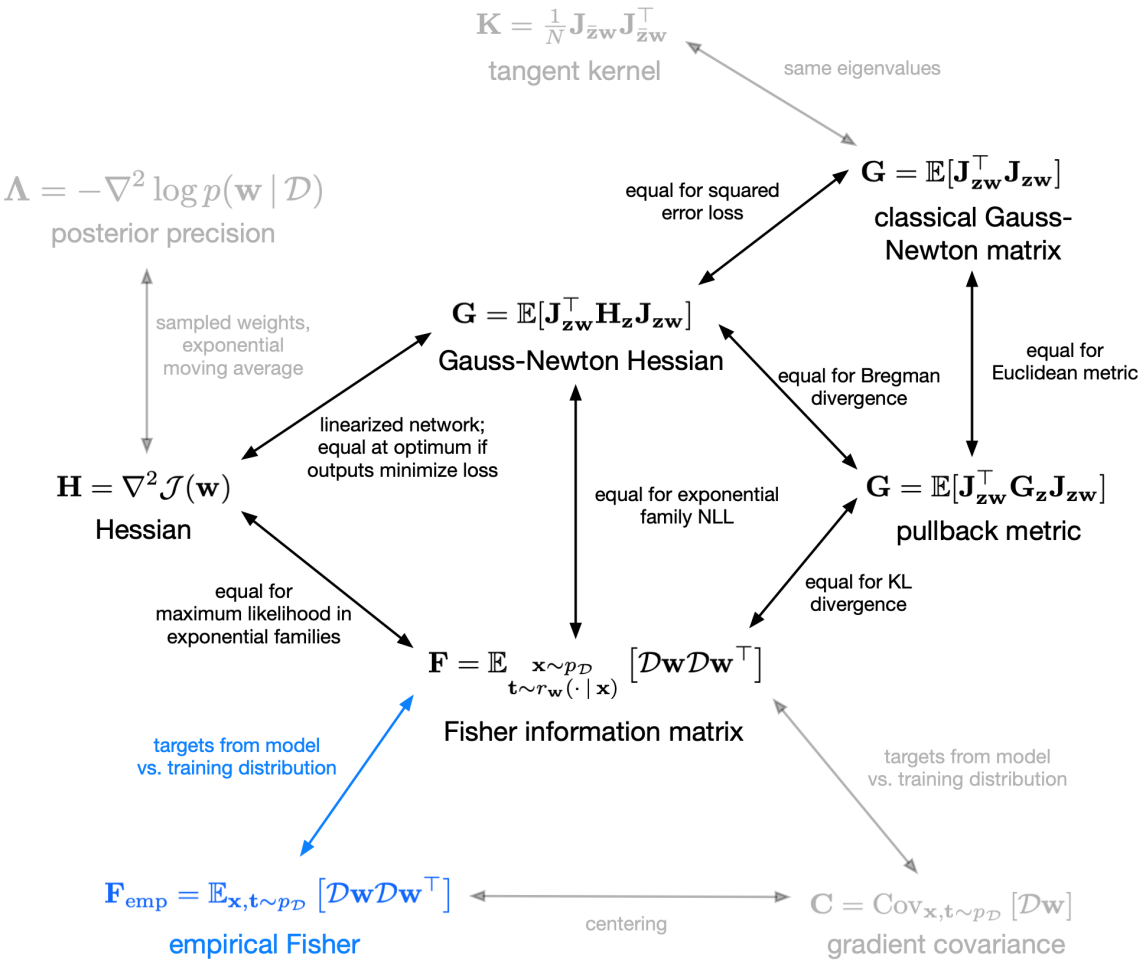

Figure 1: A summary of the relationships between the matrices used in this course. New items are in blue, and items yet to be covered are grayed out.

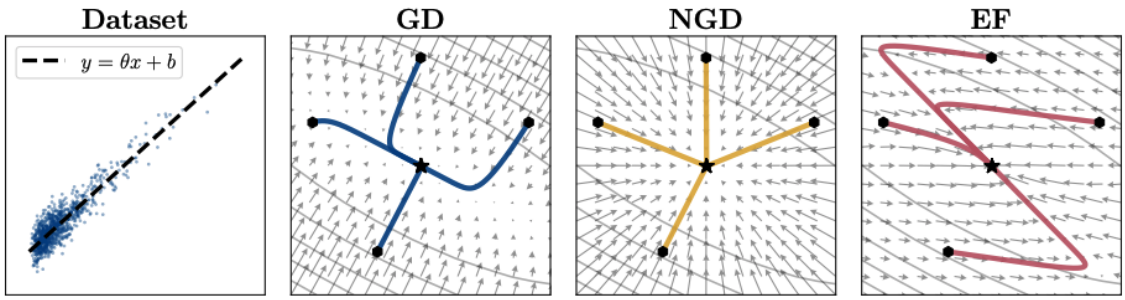

Figure 2: From Kunstner et al. (2019). Comparison of optimization paths of gradient descent, natural gradient descent, and empirical-Fisher-preconditioned gradient descent for a deterministic least squares problem (fitting the regression dataset shown on the left).

label noise, and the predictions are well calibrated (such that the training labels behave as if they were sampled from the model's output distribution). If these conditions are not satisfied, then $\mathbf{F}_{\text{emp}}$ can behave very differently from $\mathbf{H}$. For instance, consider the extreme case of the full-batch gradient (so that the gradient covariance is 0 and therefore $\mathbf{F}_{\text{emp}}$ is an exponential moving average of $\nabla \mathcal{J}(\mathbf{w}) \nabla \mathcal{J}(\mathbf{w})^\top$), and imagine we substitute in $\mathbf{F}_{\text{emp}}$ for $\mathbf{H}$ in the Newton update:

$$\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k-1)} - \mathbf{F}_{\text{emp}}^{-1} \nabla \mathcal{J}(\mathbf{w}^{(k-1)}). \tag{3}$$

Roughly speaking, we can see that $\mathbf{F}_{\text{emp}}$ scales *quadratically* in the size of the gradient. Therefore, the algorithm will take a *smaller* step in directions that consistently have a *larger* gradient signal. This feels like an undesirable property from an optimization standpoint, and indeed, Figure 2 shows that preconditioning using $\mathbf{F}_{\text{emp}}^{-1}$ results in a rather bizarre path when applied to a simple least-squares problem. This effect is why algorithms based on the empirical Fisher matrix (such as RMSprop, Adam, etc.) nearly always precondition using $\mathbf{F}_{\text{emp}}^{-1/2}$ rather than $\mathbf{F}_{\text{emp}}^{-1}$. See Kunstner et al. (2019) for a more detailed discussion of the problems with the empirical Fisher matrix when it comes to optimization.

The defition of $\mathbf{F}_{\text{emp}}$ above assumed a batch size of 1. In practice, like SGD, adaptive gradient methods are typically run on batches of data. In principle, the proper thing to do would be to find a way to estimate Eqn. 2 in a way that uses batch operations. Unfortunately, this is hard to do in practice, so instead one simply substitutes the batch gradients for the per-example gradients in Eqn. 2. Letting $\mathcal{B}$ denote a training batch, the empirical Fisher matrix becomes:

$$\begin{aligned} \mathbf{F}_{\text{emp}}^{\text{batch}} &= \mathbb{E}_{\mathcal{B} \sim p_{\text{data}}} [\nabla \mathcal{J}_{\mathcal{B}}(\mathbf{w}) \nabla \mathcal{J}_{\mathcal{B}}(\mathbf{w})^\top] \\ &= \text{Cov}_{\mathcal{B} \sim p_{\text{data}}} (\nabla \mathcal{J}_{\mathcal{B}}(\mathbf{w})) + \nabla \mathcal{J}(\mathbf{w}) \nabla \mathcal{J}(\mathbf{w})^\top, \\ &= \frac{1}{|\mathcal{B}|} \text{Cov}_{(\mathbf{x},\mathbf{t}) \sim p_{\text{data}}} (\nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})) + \nabla \mathcal{J}(\mathbf{w}) \nabla \mathcal{J}(\mathbf{w})^\top. \end{aligned} \tag{4}$$

So while the true empirical Fisher matrix $\mathbf{F}_{\text{emp}}$ is independent of $|\mathcal{B}|$, the version computed in practice depends on $|\mathcal{B}|$. As $|\mathcal{B}|$ gets larger, $\mathbf{F}_{\text{emp}}^{\text{batch}}$ gets smaller. This means larger batch sizes imply larger effective learning

rates, a surprising gotcha when tuning hyperparameters (and one not shared by SGD). Also, the covariance term shrinks relative to the gradient outer product term, causing it to less resemble the true Fisher matrix.

The original Adam paper is one of the most highly cited scientific papers ever, and there have been dozens of follow-up papers proposing minor variations on the algorithm. Given that there are such profound differences between the empirical and true Fisher matrices, as well as between the per-example and per-batch versions of the empirical Fisher, you'd expect someone to have measured the effects of these various choices on optimization performance. But to the best of my knowledge, this hasn't been done. Good opportunity for a class project!

2.2 Why are diagonal approximations effective?

In Chapter 4, we considered Kronecker-factored approximations to neural net curvature. But in practice, the most widely used adaptive gradient methods all use the much cruder diagonal approximations. This isn't inevitable, e.g. Shampoo (Gupta et al., 2018) is an adaptive gradient method which uses a K-FAC-like factorization. But why have diagonal preconditioners been so successful? There isn't *that* much to distinguish individual coordinates in neural net training, so we wouldn't expect the eigenvectors of $\mathbf{H}$, $\mathbf{G}$, etc. to be well-aligned with the coordinate axes.

Agarwal et al. (2020) argue that what's actually important isn't so much *diagonal* rescaling as *layerwise* rescaling. Different layers in a network play different computational roles, for instance because some feed into normalization layers and others don't (see Section 4). The gradients will have different scales in different layers, and it's useful to be able to correct for that. Because each parameter belongs to one layer, diagonal rescaling will be able to capture the effect of layerwise rescaling. It's questionable whether diagonal rescaling is doing much more than this. E.g., Agarwal et al. (2020) run experiments where they modify SGD updates with layerwise rescaling so that each layer's scale matches that of Adam (or vice versa). For a variety of benchmarks, this simple manipulation closes the performance gap between algorithms, suggesting that more fine-grained diagonal scaling of the gradients within a layer isn't very important for optimization.

Normalization operations can have a dramatic effect on the scales of gradients for individual layers; see Section 4.3.

3 The Problem of Internal Covariate Shift

All the way back in Chapter 1, we analyzed the gradient descent dynamics for linear regression. Since the cost function is quadratic, the convergence rate depends on the condition number of the Hessian, which is given by

$$\mathbf{H} = \frac{1}{N} \sum_{i=1}^{N} \breve{\mathbf{\Phi}}^\top \breve{\mathbf{\Phi}},$$

where $\breve{\mathbf{\Phi}}$ is the design matrix with a homogeneous coordinate. We observed that if the features are uncentered, i.e. $\mathbb{E}[\boldsymbol{\phi}(\mathbf{x})] = \mathbf{m} \neq \mathbf{0}$, then $\mathbf{H}$ has a large outlier eigenvalue of magnitude $1 + \|\mathbf{m}\|^2$; this is an outlier because $\|\mathbf{m}\|^2$ grows linearly in the dimension. Another way that ill-conditioning can arise is if all the features have different variances. In the case of linear

regression, there was a straightforward fix: normalize the inputs to be zero-mean and unit variance. This does not guarantee that the optimization problem will be well-conditioned, but it fixes the specific aforementioned source of ill-conditioning.

A fully connected layer of a neural network computes a linear function of the previous layer's activations $\mathbf{a}_{\ell-1}$, followed by a nonlinear activation function. Since the linear part resembles linear regression, it's a reasonable guess that the Hessian with respect to the weights $\mathbf{W}_\ell$ will be better conditioned if $\mathbf{a}_{\ell-1}$ has zero mean and unit variance. Unfortunately, unlike in the linear regression case, there is no straightforward way to normalize the activations, since they are computed from a learned feature map. Even if the network is somehow initialized so that the activations have zero-mean and unit variance, the statistics will change as the network trains, a problem referred to as **internal covariate shift (ICS)**. This is not a new observation: it was made by LeCun et al. (1991), only a few years after backprop was invented (though the term ICS is more recent).

We can actually make a more precise prediction about the effect of ICS on optimization. Recall from Chapter 4 that the K-FAC approximation to the Gauss-Newton Hessian for layer ℓ is given by:

$$\hat{\mathbf{G}}_{\ell\ell} = \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell, \tag{5}$$

where $\mathbf{A}_{\ell-1}$ and $\mathbf{S}_\ell$ are the (uncentered) covariances of the activations and pre-activation pseudo-gradients, respectively. Since $\mathbf{A}_{\ell-1}$ is defined exactly analogously to $\mathbf{H}$ in the linear regression problem, all of our conclusions about the eigenvalues of $\mathbf{H}$ also apply directly to the eigenvalues of $\mathbf{A}_{\ell-1}$. In particular, if $\mathbf{a}_{\ell-1}$ is uncentered, then we'd expect $\mathbf{A}_{\ell-1}$ to have a single large outlier eigenvalue (call it ν). Observe now that the eigenvalues of a Kronecker product are the products of the eigenvalues of the two factors. This leads to the following prediction about the conditioning of $\mathbf{G}_{\ell\ell}$:

> **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 $\mathbf{v}$.

The above argument is only heuristic, as K-FAC is only an approximation. Does it hold up in practice? This is actually pretty hard to answer, since it's hard to obtain reliable information about the eigenspectra of neural net Hessians. Even though theories of ICS and conditioning are decades old, we've only recently started to see concrete evidence. Ghorbani et al. (2019) found evidence for a small number of outlier eigenvalues which disappear when batch norm is applied. Papyan (2020) performed a more rigorous mathematical analysis of the spectra of neural net Hessians and reached a similar conclusion, namely that there are a small number of outlier eigenvalues related to the un-centering of the activations. This conclusion was backed up with extensive experimental evidence. Therefore, I feel comfortable asserting that ICS exerts a significant effect on the conditioning of neural net Hessians.

Papyan (2020) found C large eigenvalues (where C is the number of classes), rather than M, as claimed above. You could probably argue that if the network is close to linear, then $\mathbf{S}_\ell$ will be close to rank-C, and therefore you get C large eigenvalues.

3.1 Changing One Thing at at Time: Preconditioning

The above analysis was indeed one of the main motivations behind batch norm, but it's important to remember that batch norm wasn't the *first* normalization scheme for neural nets. Rather, it followed on a long line of other normalization schemes, and was designed to achieve other goals *in addition* to improving the conditioning. A good scientist changes only one thing at a time, so let's consider how one might directly fix the ill-conditioning caused by ICS while changing as little as possible about the training.

The following analysis is fairly representative of predecessors to batch norm, but isn't intended to exactly match any particular algorithm.

The way to do this is with preconditioning. Recall from Chapter 4 that preconditioning refers to transforming to another coordinate system where the Hessian is better conditioned. Suppose we have a fully connected layer that computes:

$$\mathbf{a}_\ell = \phi(\mathbf{W}_\ell \mathbf{a}_{\ell-1} + \mathbf{b}_\ell). \tag{6}$$

We'd like to normalize the activations using estimates of the first- and second-order statistics, for instance estimated using exponential moving averages. Let $\mathbf{m}$ be the estimated mean and $\mathbf{\Sigma}$ be a diagonal matrix containing the estimated variances. The normalized activations are defined as:

$$\tilde{\mathbf{a}}_{\ell-1} = \mathbf{\Sigma}^{-1/2}(\mathbf{a}_{\ell-1} - \mathbf{m}). \tag{7}$$

We'd like to choose weights and biases to ensure that the network still computes the same function, i.e.

$$\tilde{\mathbf{W}}_\ell \tilde{\mathbf{a}}_{\ell-1} + \tilde{\mathbf{b}}_\ell = \mathbf{W}_\ell \mathbf{a}_{\ell-1} + \mathbf{b}_\ell, \tag{8}$$

and this is achieved by:

$$\tilde{\mathbf{W}}_\ell = \mathbf{W}_\ell \mathbf{\Sigma}^{1/2} \qquad \tilde{\mathbf{b}}_\ell = \mathbf{b}_\ell + \mathbf{W}_\ell \mathbf{m}. \tag{9}$$

In the preconditioned space, $\mathbf{m} = \mathbf{0}$, so under the Conditioning Hypothesis, we'd expect this preconditioning to eliminate the outlier eigenvalues.

The transformation we just defined is a linear reparameterization, so it can be analyzed using the analysis of Chapter 4. Defining $\mathbf{w}_\ell = (\mathrm{vec}(\mathbf{w}_\ell)^\top \, \mathbf{b}_\ell^\top)^\top$, the above transformation can be written as $\tilde{\mathbf{w}}_\ell = \mathbf{R}^{-1}\mathbf{w}_\ell$, and the update rule is equivalent to pre-multiplying the gradient by $\mathbf{R}\mathbf{R}^\top$. The formulas are:

$$\mathbf{R}^{-1} = \begin{pmatrix} \mathbf{\Sigma}^{1/2} \otimes \mathbf{I} & \mathbf{0} \\ \mathbf{m}^\top \otimes \mathbf{I} & 1 \end{pmatrix} \qquad [\mathbf{R}\mathbf{R}^\top]^{-1} = \begin{pmatrix} (\mathbf{\Sigma} + \mathbf{m}\mathbf{m}^\top) \otimes \mathbf{I} & \mathbf{m} \otimes \mathbf{I} \\ \mathbf{m}^\top \otimes \mathbf{I} & 1 \end{pmatrix} \tag{10}$$

Recall that $[\mathbf{R}\mathbf{R}^\top]^{-1}$ is intended to approximate the Hessian. By inspecting the right-hand formula, we see that this matrix is *almost* diagonal, except that it has additional terms to capture the off-centering. For this reason, preconditioners of this form are known as **quasi-diagonal** preconditioners (Ollivier, 2015).

It's interesting to contrast this with diagonal preconditioners such as the adaptive gradient methods. If the activations are centered ($\mathbf{m} = \mathbf{0}$), then the preconditioner in Eqn. 10 is diagonal. Hence, a diagonal preconditioner is able to compensate for the effect of different activations having different

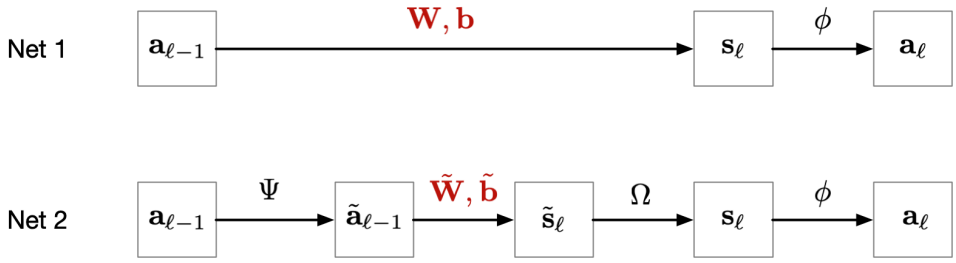

Figure 3: Illustration of the desired affine invariance property. **(top)** A layer of an ordinary MLP. **(bottom)** The same architecture, but where invertible affine transformations $\mathbf{\Psi}$ and $\mathbf{\Omega}$ are applied to the activations and preactivations. Trainable parameters are shown in red. The parameters $\tilde{\mathbf{W}}$ and $\tilde{\mathbf{b}}$ can be chosen such that Net 2 computes the same function as Net 1, in which case Net 2 is simply a reparameterization of Net 1. After an SGD update, the two networks will compute different functions, while after an (undamped) K-FAC update, they will continue to compute the same function. This is what is meant by invariance to affine reparameterization.

scales. However, the effect of off-centering can't be captured by a diagonal matrix. (For example, suppose $\mathbf{\Sigma} = \mathbf{I}$ and $\mathbf{m}$ is a vector all of whose entries are 0.1. In this case, we still get outlier eigenvalues of $0.1M$, but this is hardly reflected in the diagonal entries, which all have the value 1.1.) To compensate for off-centering, we need (at least) the quasi-diagonal approximation. Fortunately, quasi-diagonal preconditioners aren't much more expensive than diagonal ones (since they can be computed with Eqn. 9). This is an important insight about neural net training, and hardly anyone knows about it today!

3.2 Invariance

An alternative way to arrive at quasi-diagonal preconditioners is by reasoning about invariance. Recall that natural gradient descent was motivated by invariance to smooth reparameterizations: if we reparameterize the weights of our network using a smooth transformation, then the natural gradient updates in both parameterizations will be equivalent, up to the first order, in terms of the predictions made by the network.

Computing the exact natural gradient is generally impractical. It's also pretty extravagant to ask for invariance to *arbitrary* smooth reparameterizations, since most of these are nonsensical. Fundamentally, a fully connected layer of a neural net computes an affine mapping between two affine spaces. To parameterize an affine mapping, it suffices to specify affine bases for each of the two spaces. There are, of course, other ways to parameterize the mapping; for instance, we can randomly permute all of the entries of its matrix representation. But such parameterizations are unnatural. By limiting ourselves to the natural reparameterizations, we can achieve invariance much more efficiently than by computing the exact natural gradient.

Figure 3 illustrates the sort of affine reparameterizations we're talking about. In each layer, the activations are passed through an affine transformation (which can be seen as an affine change-of-basis), which for the purposes of this analysis is assumed to be fixed. The FC layer computations

are applied to get the transformed pre-activations. Then we apply another affine transformation, which we also assume to be fixed. We'd like to make this entire computation equivalent to that of the original network, and this can be achieved with a particular setting of the transformed parameters.

With a bit of drudgery, it can be shown that (undamped) K-FAC is invariant to this sort of affine reparameterization. Interestingly, unlike the generic result for natural gradient descent, this invariance property holds *exactly*. This property was shown using elementary linear algebra by Martens and Grosse (2015), and later Luk and Grosse (2018) showed how to construct K-FAC out of coordinate-free mathematical objects so that the invariance property follows automatically.

Now suppose we further limit the set of allowable transformations to *coordinatewise* affine transformations of the activations and pre-activations. I.e., we can apply arbitrary affine transformations to individual units independently. By designing an algorithm to be invariant to this class of transformations, we arrive at a quasi-diagonal preconditioner analogous to Eqn. 10. Such a quasi-diagonal preconditioner was derived by Ollivier (2015).

4 Batch Normalization

The above discussion of the ill-conditioning effects of ICS and their solution using preconditioning sets the stage for our discussion of batch normalization (BN) (Ioffe and Szegedy, 2015). This is an operation which linearly transforms the columns of a matrix (typically representing the activations $\mathbf{A}_\ell$ or pre-activations $\mathbf{S}_\ell$ for a batch) to have zero mean and unit variance. The operation is defined as follows:

$$\tilde{\mathbf{X}} = \mathrm{BN}(\mathbf{X}) = (\mathbf{X} - \mathbf{1}\boldsymbol{\mu}(\mathbf{X})^\top) \oslash \mathbf{1}\boldsymbol{\sigma}(\mathbf{X})^\top, \tag{11}$$

In practice, the BN operation includes learnable parameters for the output mean and variance for each column. This is done in order that BN maintain the expressive power of the original network. For simplicity, we assume these are fixed at mean 0 and standard deviation 1. I believe this doesn't significantly impact any of the analyses here.

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are functions which compute the columnwise mean and standard deviation, respectively, and $\oslash$ denotes elementwise division. Note that the mean and variance statistics are estimated from the *current* batch, i.e. we're not keeping moving averages. Typically, the BN operation is applied to the pre-activations in each layer, although it is possible to apply it to the activations instead.

What distinguishes BN from the preconditioning approach described above is that BN is treated as a layer in the network. Hence, it's included in the computation graph, and we include its VJP when doing backprop. BN should be thought of as a modification to the network architecture, rather than as an optimization algorithm. Figure 4 shows the computation graph for BN, which operates on batches rather than individual examples. It's convenient to divide this graph into two distinct paths, a **direct path** and a **statistics path**, so that the contribution of each path to the backprop computation can be analyzed separately. The direct path can be thought of as the backprop computation when the statistics are assumed to be fixed. It's not hard to show that this consists of a simple rescaling:

In this sense, BN is to be contrasted with the preconditioning approach. Preconditioning keeps the network's forward and backprop computations the same, and can be thought of as a linear transformation of the completed gradient.

$$\overline{\mathbf{X}} = \overline{\tilde{\mathbf{X}}} \oslash \mathbf{1}\boldsymbol{\sigma}(\mathbf{X})^\top \tag{12}$$

The bar denotes backpropagated derivatives. See the Backprop lecture notes for CSC413.

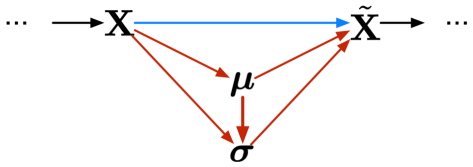

Figure 4: Computation graph for the batch normalization operation. The direct path is shown in blue, and the statistics path in red.

The statistics path accounts for the fact that changing $\mathbf{X}$ changes the statistics, and the formulas are more cumbersome. We'll see that both paths play important roles in the optimization effects of BN.

4.1 Motivations

BN was indeed motivated partly as a way to combat the ill-conditioning effects of ICS. However, it was designed to provide *additional* benefits at the same time, including:

- preventing dead or saturated units
- maintaining stability at higher learning rates
- a regularization effect resulting from the normalizer being stochastic, depending on which other training examples are selected for the batch

All of these effects can be reasoned through from first principles, and I'll discuss them each briefly. The BN paper didn't attempt to disentangle which of these factors are important for performance, but you can probably imagine how you would design experiments to test them. In addition to the intended effects of BN, there is also a very important phenomenon not addressed by the original paper, namely an implicit learning rate decay. This is discussed in Section 4.3.

The first effect — preventing dead or saturated units — is straightforward. If the inputs to the ReLU activation function are always 0, then that unit is dead: its value is always zero, so it has no effect on the network's computations. Furthermore, the incoming weights receive a gradient signal of 0, so there is nothing encouraging the unit to come back to life. However, if the pre-activations are forced by BN to be 0 in expectation, then they will be positive some of the time, and the unit won't be dead. Analogous arguments apply to other activation functions.

Despite the zero gradient, it is possible for the unit to come back to life if updates to other parts of the network cause the preceding layer's activations to change in a way such that the unit's inputs become positive again.

Stochastic regularization will be discussed in Chapter 8.

4.2 A Wrinkle in the ICS Story

The analysis of Section 3 identified a problem in need of a solution: uncentered activations create some outlier eigenvalues in the Hessian or Fisher information matrix, each one of the form $\mathbf{m} \otimes \mathbf{v}$ for some vector $\mathbf{v}$. In Section 3.1, we saw that these outlier eigenvalues could be removed, without changing the expressive power of the model, by transforming the activations to have mean zero. Since BN explicitly transforms the activations to have

mean zero (within a batch), this suggests a natural hypothesis for how BN improves optimization:

Again, we are ignoring the mean and gain parameters of BN for simplicity.

> **ICS Removal Hypothesis.** BN improves optimization by centering and/or normalizing the previous layer's activations.

Note that the ICS Removal Hypothesis is logically independent of the ICS Conditioning Hypothesis. It could be that BN removes the outlier eigenvalues by some means other than centering the previous layer's activations, or conversely it could be that centering the previous layer's activations helps optimization for some reason other than eliminating the outlier eigenvalues.

The ICS Removal Hypothesis seems pretty reasonable in light of the above discussion, but it's contradicted by two pieces of evidence. First of all, observe that we have a choice of whether to apply BN before or after the activation function. BN centers the activations only if it's applied after the activation function, so the ICS Removal Hypothesis implies this is what ought to be done. In fact, ReLU always outputs a positive activation, so the activations are *guaranteed* to be uncentered if BN is applied before the activation. But in practice, it's more common to apply it *before* the activation function, and indeed the inventors tried both methods and found that applying it before the activation function generally worked better. This is hard to square with the ICS Removal Hypothesis.

Another piece of evidence comes from a neat experiment by Santurkar et al. (2018). Essentially, it is a sort of "knockout" experiment where we knock out the ICS removal effect of BN and see what is the effect on optimization. If the ICS Removal Hypothesis is correct, we'd expect the resulting performance to degrade to the level without BN. The following is a simplification of their manipulation, but I think it captures the main idea. Basically, for each batch, we transform the activations in a given layer ℓ using a random affine transformation, so that the first- and second-order statistics are changed, but information is otherwise preserved:

Here, i indexes the training example within the batch, and j indexes the hidden unit. The layer index ℓ is omitted for clarity.

For $j = 1, \dots, D$

$$m_j \sim p_m$$
$$\gamma_j \sim p_\gamma$$

For $i = 1, \dots, B$

For $j = 1, \dots, D$

$$\hat{a}_j^{(i)} = \gamma_j a_j^{(i)} + m_j$$

The distrubtions p_m and p_s can be chosen such that the distributions of means and variances resemble those seen during non-BN training. Performing this random transformation after a BN layer can re-introduce the ICS that was removed by BN, and therefore (seemingly) re-introduce the outlier eigenvalues that BN removed. The main observation of Santurkar et al. (2018) was that this transformation has almost no effect on the optimization performance, suggesting that ICS removal is not the real mechanism.

So do we need to throw out the whole ICS story? Not so fast. The key here is to notice what happens when BN is applied in the *next* layer, before

the activation function. In other words, we're interested in the following sequence of computations:

Here we're assuming a fully connected layer, but similar arguments apply to other layer types. Note that there is no bias term in the linear part because BN undoes the effect of the bias anyway.

$$\begin{aligned}\mathbf{R}_\ell &= \mathbf{A}_{\ell-1}\mathbf{W}_\ell^\top \\ \mathbf{S}_\ell &= \mathrm{BN}(\mathbf{R}_\ell) \\ \mathbf{A}_\ell &= \phi_\ell(\mathbf{S}_\ell)\end{aligned}$$

Let's absorb this all into a single function f_ℓ:

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

The important thing to observe is that this function is invariant to shifting and rescaling of the activations:

Note that this invariance result only applies to scalar multiplication by s, rather than to unitwise rescaling.

$$f_\ell(\mathbf{A}_{\ell-1}, \mathbf{W}_\ell) = f_\ell(s\mathbf{A}_{\ell-1} + \mathbf{1}\mathbf{m}^\top, \mathbf{W}_\ell) \qquad \text{for any } s, \mathbf{m}$$

If the network's function is invariant to rescaling and shifting $\mathbf{a}_{\ell-1}$, then so is the cost function $\mathcal{J}(\mathbf{W}_\ell)$, and therefore also its Hessian, Fisher matrix, etc. Hence, the conditioning of the cost function is independent of the centering and scaling of $\mathbf{a}_{\ell-1}$. We'd expect the outlier eigenvalues to disappear, just like with a quasi-diagonal preconditioner. So the ICS Conditioning Hypothesis is still looking pretty solid, except that the ill-conditioning is fixed by applying BN to the *next* layer, not the *previous* one.

The explanation given here is inspired by a conversation with Sergey Ioffe and Ian Goodfellow. I haven't been able to find a published paper that discusses it. Rigorously testing this explanation could make a good class project.

An alternative way to view the same phenomenon is in terms of the mechanics of the backprop computations. For clarity, let's focus on the batch centering (BC) operator, which centers but doesn't rescale the activations:

$$\mathbf{S} = BC(\mathbf{R}) = \mathbf{R} - \mathbf{1}\boldsymbol{\mu}(\mathbf{R})^\top. \tag{13}$$

The backprop computations for this operator can be decomposed into a direct path and a statistics path (see Figure 4):

Interestingly, backpropping through the BC operator has the effect of centering the gradients.

$$\overline{\mathbf{R}} = \underbrace{\overline{\mathbf{S}}}_{\text{direct path}} - \underbrace{\mathbf{1}\boldsymbol{\mu}(\overline{\mathbf{S}})^\top}_{\text{statistics path}}. \tag{14}$$

Let $\mathbf{A}$ denote the previous layer's activations, and decompose it into the centered activations $\tilde{\mathbf{A}}$ and mean $\mathbf{m}$ as follows:

$$\mathbf{A} = \tilde{\mathbf{A}} + \mathbf{1}\mathbf{m}^\top.$$

The batch gradient is computed as follows:

In the second-to-last step, $\mathbf{1}^\top\tilde{\mathbf{A}} = \mathbf{0}$ because $\tilde{\mathbf{A}}$ is centered. Similarly, $(\overline{\mathbf{S}} - \mathbf{1}\boldsymbol{\mu}(\overline{\mathbf{S}})^\top)^\top\mathbf{1} = \mathbf{0}$ because $\overline{\mathbf{S}} - \mathbf{1}\boldsymbol{\mu}(\overline{\mathbf{S}})^\top$ is centered.

$$\begin{aligned}\overline{\mathbf{W}} &= \overline{\mathbf{R}}^\top\mathbf{A} \\ &= \left(\overline{\mathbf{S}} - \mathbf{1}\boldsymbol{\mu}(\overline{\tilde{\mathbf{S}}})^\top\right)\left(\tilde{\mathbf{A}} + \mathbf{1}\mathbf{m}^\top\right) \\ &= \big[\overline{\mathbf{S}}^\top\tilde{\mathbf{A}} - \underbrace{\boldsymbol{\mu}(\overline{\mathbf{S}})\mathbf{1}^\top\tilde{\mathbf{A}}}_{=\mathbf{0}} + \underbrace{(\overline{\mathbf{S}} - \mathbf{1}\boldsymbol{\mu}(\overline{\mathbf{S}})^\top)^\top\mathbf{1}\mathbf{m}^\top}_{=\mathbf{0}}\big] \\ &= \overline{\mathbf{S}}^\top\tilde{\mathbf{A}}\end{aligned}$$

But this last formula is essentially the weight gradient computed using the *centered* activations! Hence, off-centering has no impact on the dynamics,

and we'd expect the outlier eigenvalues to disappear. Note that this happens regardless of whether or not we use BN (or some other method) to center the previous layer's activations!

It's interesting that the statistics path plays such an important role in making the gradients more well-behaved. What's going on? Recall that the outlier eigenvectors are of the form $\mathbf{v} \otimes \mathbf{m}$. It can be shown that adjusting the weights in this direction causes the next layer's pre-activations to be changed by approximately a multiple of $\mathbf{1}\mathbf{v}^\top$. In other words, the pre-activations of every example are transformed additively in exactly the same way. This isn't a very useful change, because it doesn't help distinguish the training examples from each other. When BN is applied, this change is undone by BN's centering operation, which cancels out any change to the mean pre-activations. I.e., BN filters out changes to the weights which don't help distinguish the training examples, allowing the network to focus on updating the weights in directions which are more meaningful.

4.3 Implicit Learning Rate Decay

Improving the conditioning of the cost function isn't the whole story for how BN improves optimization. There is a very important side effect of BN (and other normalization schemes, such as weight norm and layer norm), namely that it causes an implicit learning rate decay effect.

This section is loosely based on van Laarhoven (2017); Hoffer et al. (2017); Zhang et al. (2019); Li and Arora (2020); Roburin et al. (2020).

A classical result will help us out here. A function g is said to be **homogeneous of degree** k if

$$g(\gamma \mathbf{x}) = \gamma^k g(\mathbf{x}) \tag{15}$$

for any $\mathbf{x}$. If this holds for $k = 0$, it is said to be **scale-invariant**. In the case that g is scalar-valued, **Euler's homogeneous function theorem** tells us that

$$\mathbf{x}^\top \nabla g(\mathbf{x}) = k g(\mathbf{x}). \tag{16}$$

In particular, if g is scale-invariant, then $\mathbf{x} \perp \nabla g(\mathbf{x})$. A corollary of this result is that the gradient ∇g is homogeneous of degree $k - 1$, i.e.

The symbol $\perp$ denotes orthogonality.

$$\nabla g(\gamma \mathbf{x}) = \gamma^{k-1} \nabla g(\mathbf{x}). \tag{17}$$

Now observe that the BN operation is invariant to rescaling the incoming weights to a given unit. Mathematically,

Here, we continue to assume BN is applied *before* the activation function. The function f_ℓ denotes the layer's computations, as in Section 4.2.

$$f_\ell(\mathbf{A}, \mathbf{W}) = f_\ell(\mathbf{A}, \mathbf{\Gamma W}) \qquad \text{for any diagonal matrix } \mathbf{\Gamma} \succ \mathbf{0}.$$

This implies the layer's computations are scale invariant when viewed as a function of $\mathbf{w}_j$, the vector of incoming weights to unit j. Consider $\mathcal{J}(\mathbf{w}_j)$, the cost viewed as a function of $\mathbf{w}_j$. This is the composition of f_ℓ with the rest of the network's computations and the loss function. Hence, $\mathcal{J}(\mathbf{w}_j)$ is also scale invariant. Consequently, by Eqn. 17, the gradient is homogeneous of degree -1:

$$\nabla \mathcal{J}(\gamma \mathbf{w}_j) = \gamma^{-1} \nabla \mathcal{J}(\mathbf{w}_j). \tag{18}$$

Since the network's function and the cost are scale invariant, the scale of $\mathbf{w}_j$ doesn't matter, hence we can understand the dynamics by canonicalizing

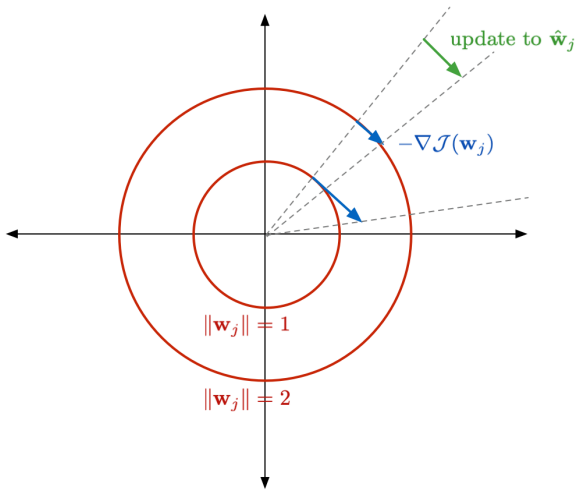

Figure 5: Illustration of batch normalization's implicit learning rate decay effect. The gradient $\nabla\mathcal{J}(\mathbf{w}_j)$ is tangent to the sphere centered at the origin, and its norm scales inversely to $\|\mathbf{w}_j\|$. Because the gradients are tangent to the sphere, the norm of the weights grows over time.

$\mathbf{w}_j$ to a unit vector $\hat{\mathbf{w}}_j = \mathbf{w}_j/\|\mathbf{w}_j\|$. Observe that the change to $\hat{\mathbf{w}}_j$ resulting from an SGD upate can be approximated as:

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

This equation can be interpreted in two ways. First, the vector $\|\mathbf{w}_j^{(k)}\|^{-2}\nabla\mathcal{J}(\hat{\mathbf{w}}_j^{(k)})$ can be considered the **effective gradient**, since it's the vector that would be plugged into an imaginary SGD update to $\hat{\mathbf{w}}_j$. Alternatively, $\hat{\alpha} = \alpha\|\mathbf{w}_j^{(k)}\|^{-2}$ can be considered the **effective learning rate**. The above derivation shows that the effective gradient and the effective learning rate are each homogeneous of degree -2, meaning that scaling the weights by γ results in the size of the update being scaled by γ^{-2}. This effect is illustrated in Figure 5.

The next thing to notice is that the norm $\|\mathbf{w}_j\|$ increases monotonically. This can be understood geometrically as follows (see also Figure 5): plugging in $k = 0$ to Eqn. 16, we find that $\mathbf{w}_j \perp \nabla\mathcal{J}(\mathbf{w}_j)$ for all $\mathbf{w}_j$. By the

Ask yourself: is it the direct path or the statistics path which is responsible for the implicit learning rate effect?

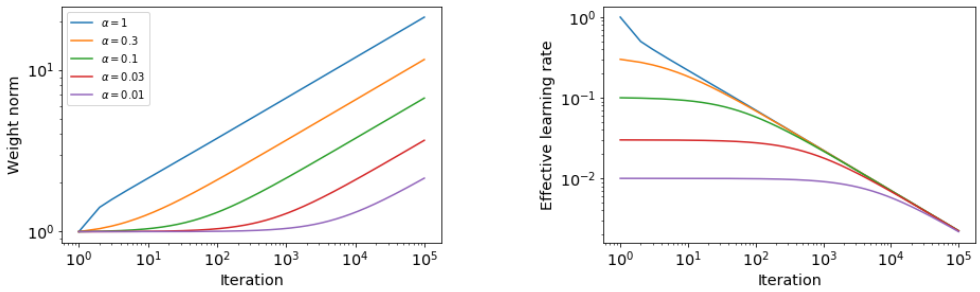

Figure 6: **(left)** Growth of the norm $\|\mathbf{w}_j^{(k)}\|$ for various explicit learning rates α, assuming $\|\mathbf{w}_j^{(0)}\| = 1$ and $\|\nabla\mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\| = 1$ for all k. **(right)** The corresponding effective learning rate schedules. Note the logarithmic scales on both axes.

Pythagorean Theorem, the norm of the weights increases according to:

$$\begin{aligned}\|\mathbf{w}_j^{(k+1)}\|^2 &= \|\mathbf{w}_j^{(k)} + \alpha\nabla\mathcal{J}(\mathbf{w}_j^{(k)})\|^2 \\ &= \|\mathbf{w}_j^{(k)}\|^2 + \alpha^2\|\nabla\mathcal{J}(\mathbf{w}_j^{(k)})\|^2 \\ &= \|\mathbf{w}_j^{(k)}\|^2 + \frac{\alpha^2\|\nabla\mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\|^2}{\|\mathbf{w}_j^{(k)}\|^2}\end{aligned} \tag{19}$$

The exact pattern of growth depends on $\nabla\mathcal{J}(\hat{\mathbf{w}}_j^{(k)})$, which is hard to characterize in general. However, to gain intuition, we can suppose $\|\nabla\mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\|$ is a constant. In this case, it can be shown that the norm grows approximately as:

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

This translates into a schedule for the effective learning rate $\hat{\alpha}$:

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

Interestingly, this happens to be a learning rate schedule that's popular in stochastic optimization. For instance, it is exactly a learning rate decay schedule of this form that differentiates Adagrad from RMSprop (see Section 2).

The explicit learning rate hyperparameter gives us surprisingly little control over the learning dynamics. For instance, Figure 6 shows how $\|\mathbf{w}_j^{(k)}\|$ and $\hat{\alpha}_k$ evolve over time for different choices of the explicit learning rate α, assuming $\|\mathbf{w}_j^{(0)}\| = 1$ and $\|\nabla\mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\| = 1$ for all k. For larger α, the weight norms increase more rapidly, counteracting the larger explicit learning rate. In fact, regardless of the value of α, all the effective learning rates asymptote to the same trend. The explicit learning rate hyperparameter only matters for a transient phase at the start of training!

Can we counteract this effect and achieve a constant effective learning rate? The answer is yes, but Li and Arora (2020) show that this requires an *exponentially increasing* learning rate schedule! Alternatively, we'll see in Section 5.1 that the weight decay hyperparameter, surprisingly, acts as

Bengio (2012) said, "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." In practice, the learning rate isn't *quite* so important today, and I think the implicit decay effect is a big part of why not.

another knob we can use to tune the effective learning rate schedule. In principle, a third option would be to explicitly rescale each $\mathbf{w}_j$ to be unit norm after every iteration; this doesn't change the function computed by the network, but it ensures that the effective learning rate matches the explicit learning rate. I expect that if this became standard practice, it would eliminate a lot of tricky experimental confounds. However, it is not common to do this, and matching state-of-the-art performance using such a scheme would require re-tuning a lot of other hyperparameters, possibly including explicit learning rate schedules.

Note that the implicit decay effect is not *exactly* equivalent to an $\mathcal{O}(1/\sqrt{k})$ learning rate decay schedule, for several reasons. First of all, the above analysis assumes $\|\nabla \mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\|$ is constant, but in fact we would expect it to change over time. Secondly, different weights may have different gradient magnitudes, and therefore different effective learning rates. Finally, and perhaps most importantly, normalization usually isn't applied to all layers of a network (e.g. it's rarely applied to the network outputs). This means the implicit decay effect applies only to the layers that feed into a normalization operation, and in particular it doesn't apply to the output layer.

The fact that the effective learning rate decay doesn't apply to the output layer might help explain why neural net training often behaves linearly outside the very early phase of training. The linearity of training will be the focus of Chapter 6.

Note also that the implicit decay effect is algorithm-specific. The above discussion all assumes SGD. A similar analysis for Adam also gives an effective learning rate schedule of $\mathcal{O}(1/\sqrt{k})$. However, second-order optimization methods, including approximate second-order methods like K-FAC, are immune to the implicit decay effect. This follows because the network's function is invariant to rescaling the weights, so rescaling the weights can be viewed as an affine reparameterization of the network. The second-order methods are invariant to affine reparameterizations, so in particular they're invariant to rescaling the weights. The fact that different optimizers have different implicit decay schedules presents a problem when it comes to fairly comparing different neural net optimization algorithms.

4.4 Other Normalizers

The above discussion all focuses on batch normalization for concreteness. However, there are other normalization schemes to choose from, most notably weight normalization and layer normalization. In addition, there are dozens of variants of BN itself.

In this section, the layer indices are suppressed for clarity.

Weight normalization (Salimans and Kingma, 2016), as the name suggests, normalizes the weights. In particular, it uses a representation for each weight vector $\mathbf{w}_j$ (the vector of incoming weights to a unit) which decouples the norm and the direction:

$$\mathbf{w}_j = \mathrm{WN}(\mathbf{v}_j; g_j) = \frac{g_j}{\|\mathbf{v}_j\|}\mathbf{v}_j, \tag{20}$$

where g_j is a nonnegative scalar (the gain parameter) and $\mathbf{v}_j$ is a vector. The weights $\mathbf{w}_j$ then play their usual role in the network's computations. As with BN, WN is treated as a layer in the computation graph, and is backpropagated through in the usual way.

Layer normalization (Ba et al., 2016), as you'd expect, normalizes

the activations within each layer:

$$\begin{aligned}\hat{\mathbf{a}} &= \mathrm{LN}(\mathbf{a}; \mathbf{b}, \mathbf{g}) \\ &= \frac{\mathbf{g}}{\sigma} \odot (\mathbf{a} - \mu \mathbf{1}) + \mathbf{b} \\ \mu &= \frac{1}{D} \sum_{j=1}^{D} a_j \\ \sigma^2 &= \frac{1}{D} \sum_{j=1}^{D} (a_j - \mu)^2\end{aligned} \tag{21}$$

where $\mathbf{g}$ and $\mathbf{b}$ are gain and bias vectors. As with BN and WN, LN is treated as a layer in the computation graph, and is backpropagated through. Like BN, it can be applied before or after the nonlinear activation function, but is typically applied before.

These alternative normalizers were motivated partly by efficiency considerations or easier applicability to certain architectures such as RNNs. However, we can reason about their effects on the training dynamics, just like we did for BN.

Stochastic regularization. BN achieves a stochastic regularization effect as the result of each training example being normalized using slightly different mean and variance statistics, depending what else is in the batch. The WN and LN operations are independent of the rest of the batch, so they don't have this stochastic regularization effect.

Improved conditioning. We saw above that BN removes the outlier eigenvalues caused by uncentered activations, either by explicitly centering the activations, or by centering the backpropagated gradients (see Section 4.2). WN only includes rescaling, not centering, so it doesn't achieve this effect. Salimans and Kingma (2016) therefore recommend combining WN with the operation we refer to as *batch centering* (see Section 4.2) in order to improve the conditioning.

LN includes a centering step, but you can check that this step doesn't eliminate the outlier eigenvalues caused by ICS. Whether or not the centering step still has the effect of attenuating the outlier eigenvalues is an interesting question, and I'm not aware that it's been studied.

Implicit learning rate decay. Both WN and LN are scale invariant just like BN, so the argument from Section 4.3 holds without modification. Note that in the case of LN, scale invariance holds only at the level of the entire weight matrix, rather than the vector of incoming weights to a unit.

The implicit decay effect also applies to some normalizers that predated BN, such as the local contrast normalization layers used in the original AlexNet (Krizhevsky et al., 2012).

5 Weight Decay

Some standard tricks of neural net training are not what they appear. **Weight decay** refers to an update rule which rescales the weights of a network by a positive value less than 1:

$$\mathbf{w}^{(k+1)} \leftarrow (1 - \alpha\lambda)\mathbf{w}^{(k)}. \tag{22}$$

Here, $\mathbf{w}$ is a vector representing all of the network weights, α is the learning rate, and $\lambda \geq 0$ is a hyperparameter determining the strength of the regularization. In order for this update to make sense, we require that $\alpha\lambda < 1$.

In practice, it's common to have different WD strengths for different layers, but we assume a shared WD strength for simplicity.

When the learning algorithm is (stochastic) gradient descent, weight decay has a standard interpretation as **Tikhonov regularization**. Specifically, observe that Eqn. 22 can be seen as the gradient descent update for an L_2 **regularization** term, $\frac{\lambda}{2}\|\mathbf{w}\|^2$, which encourages solutions which are closer to the origin. This technique has been used by statisticians for decades, and its effect in regression models is well-understood. It is very common to use weight decay when training neural nets, and until recently it was believed that its benefits had to do with Tikhonov regularization.

But when it comes to neural nets, several findings cast doubt on this interpretation:

Effect on deep linear networks. For linear models, L_2 regularization has the effect of discouraging individual coefficients from being very large, while allowing moderately large coefficients. Deep linear networks can represent the same class of functions, but the effect of L_2 regularization is very different. As an intuition pump, consider a deep linear network with K layers and 1 unit per layer. The function computed by the network is $y = \beta x$, where $\beta = w_1 \cdots w_K$. You can check that for a given value of β, the regularizer is minimized when $w_1 = \cdots = w_K = \beta^{1/K}$ up to a sign flip, and it takes the value $\|\mathbf{w}\|^2 = L|\beta|^{2/K}$. Hence, L_2 regularization behaves like $L_{2/K}$ regularization, which (for $K > 2$) has a sparsifying effect: it very much prefers that the weights be exactly zero, but away from zero it has little preference.

For deep linear networks with width greater than 1, the same argument implies that L_2 regularization penalizes the Schatten quasi-norm $\|\mathbf{J_{zx}}\|_{2/K}^{2/K}$, where $\mathbf{J_{zx}}$ is the input-output Jacobian, $\|\mathbf{A}\|_p = (\sum_i \sigma_i^p)^{1/p}$, and $\{\sigma_i\}$ are the singular values of $\mathbf{A}$.

This analysis only applies exactly to linear networks, but it seems likely that its effects on networks with ReLU activations are somewhat similar. Hence, we wouldn't expect L_2 regularization of deep neural networks to behave like L_2 regularization of linear models.

Interaction with normalization. WD is commonly used for networks with normalization layers such as BN, LN, etc. The regularization interpretation of WD makes little sense in these cases, because the network's function is invariant to rescaling the weights. I.e., suppose we rescale the weights by a small constant factor. This wouldn't change the function the network computes, but we can make $\|\mathbf{w}\|^2$ arbitrarily small in this way. Hence, L_2 regularization doesn't impose any meaningful constraint on the network's capacity.

AdamW. The WD update (Eqn. 22) only corresponds to an L_2 penalty in the context of gradient descent. If one believes the regularization story, the most natural way to generalize WD to other optimization algorithms (e.g. Adam) would be to add an L_2 regularizer to the cost function, and include its gradient as part of the algorithm's usual gradient calculation. The resulting algorithm is generally not equivalent to Eqn. 22.

However, it's been observed that it can work *significantly better* to apply Eqn. 22 directly, rather than adding an L_2 regularizer, in algorithms other than gradient descent. E.g., this is the basis of the widely

used **AdamW** algorithm (Loshchilov and Hutter, 2019), and similar observations were made for K-FAC (Zhang et al., 2019). The AdamW paper didn't offer an explanation for the improvement, but offered a very thorough set of experiments which (in my opinion) demonstrated convincingly that the effect was real.

Weight decay confused my students and me for a while, because we had a hard time coming up with a hypothesis that captured its full range of effects on different architectures and training algorithms. We eventually figured out the source of our confusion: depending on the architecture and training algorithm, WD was improving the generalization performance for *at least three completely different reasons!* In some cases, it acted as a regularizer in the usual sense. In other cases, it influenced the dynamics of training. Once we realized different situations resulted from different mechanisms, we were able to pin down what was actually happening. Our findings are described in Zhang et al. (2019), and we also summarize the conclusions below.

This is an instance of a common occurrence in science: the hardest part of understanding a phenomenon can be deciding which situations belong together as instances of the same phenomenon!

5.1 Mechanism 1: Effective Learning Rates

The first WD mechanism, originally pointed out by van Laarhoven (2017), has nothing to do with constraining the model's capacity. Rather, this mechanism has to do with the influence on the effective learning rate schedule.

Suppose we are using SGD to optimize a network with BN (or some other scale-invariant architecture). As before, we'll make the simplifying assumption that $\|\nabla \mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\| = 1$ for all k. Since weight decay involves rescaling the weights by $1 - \alpha\lambda$ in each iteration, we can modify our recurrence for $\|\mathbf{w}^{(k)}\|$ (Eqn. 19) as follows:

$$\|\mathbf{w}_j^{(k+1)}\|^2 = (1 - \alpha\lambda)^2 \|\mathbf{w}_j^{(k)}\|^2 + \frac{\alpha^2 \|\nabla \mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\|^2}{\|\mathbf{w}_j^{(k)}\|^2}. \tag{23}$$

Setting both sides equal, we see that this recurrence has a fixed point of approximately

$$\|\mathbf{w}_j^{(\infty)}\|^2 \approx \sqrt{\frac{\alpha}{2\lambda}}, \tag{24}$$

resulting in an effective learning rate of

$$\hat{\alpha}_\infty = \frac{\alpha}{\|\mathbf{w}_j^{(\infty)}\|^2} \approx \sqrt{2\alpha\lambda}. \tag{25}$$

Note that without WD, i.e. $\lambda = 0$, this gives $\hat{\alpha} = 0$, consistent with the power law decay of the effective learning rate. So WD essentially gives us an implicit decay schedule which levels off to a particular value rather than continuing to decay arbitarily. A larger value of λ leads to a higher effective learning rate.

Note that this explanation doesn't fully explain *why* WD helps generalization performance, since the learning rate can have multiple effects. On one hand, increasing $\hat{\alpha}$ can have optimization benefits by improving the rate of convergence in low-curvature directions. On the other hand, learning rate

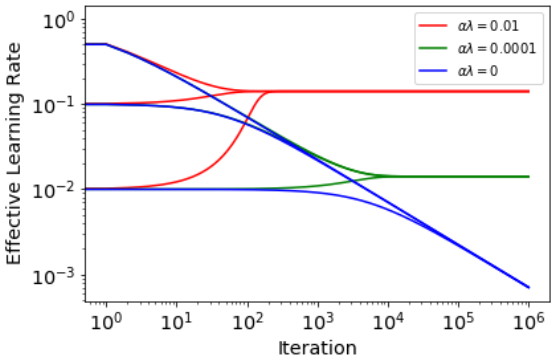

Figure 7: Effective learning rate schedule when weight decay is applied, assuming for simplicity that $\|\nabla\mathcal{J}(\hat{\mathbf{w}}_j^{(k)})\| = 1$ for all k. Note that $\hat{\alpha}$ converges to $\hat{\alpha}_\infty$ (Eqn. 25) regardless of α.

decay schedules can be important in stochastic optimization (see Chapter 7). Learning rates can also influence generalization, insofar as gradient noise can act as a regularizer (see Chapter 8). (So ironically, WD might be acting as a regularizer, just not in the way we expected!) Optimal learning rate decay schedules can vary significantly from one task and architecture to another, so it requires more work to pin down exactly what WD is doing in any particular case, even if we only consider those cases where Mechanism 1 applies.

5.2 Mechanism 2: Damping Strength

Suppose we still use an architecture with BN (or some other homogeneous normalizer) but now use the K-FAC optimizer instead of SGD. We already observed that the implicit learning rate decay effect doesn't apply to K-FAC, because K-FAC is invariant to affine reparameterizations such as rescaling the weights in a BN network. Therefore, Mechanism 1 can't explain the effectiveness of WD in training with K-FAC.

There is, however, a related explanation. Recall that the K-FAC approximation for the metric matrix $\mathbf{G}_{\ell\ell}$ for layer ℓ is defined as:

The K-FAC algorithm is discussed in detail in Chapter 4.

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

where $\mathbf{A}_{\ell-1} = \mathbb{E}[\bar{\mathbf{a}}_{\ell-1}\bar{\mathbf{a}}_{\ell-1}^\top]$ and $\mathbf{S}_\ell = \mathbb{E}[\mathcal{D}\mathbf{s}_\ell\mathcal{D}\mathbf{s}_\ell^\top]$. Recall also that for stability, when computing the natural gradient, we add a damping term to $\hat{\mathbf{G}}_{\ell\ell}$ before inversion:

In Chapter 4, we used λ to denote the damping parameter. Here we use η to avoid collision with the WD parameter λ.

$$\tilde{\nabla}\mathcal{J}(\mathbf{w}_\ell) = (\hat{\mathbf{G}}_{\ell\ell} + \eta\mathbf{I})^{-1}\nabla\mathcal{J}(\mathbf{w}_\ell). \tag{26}$$

Now, think about what happens to $\mathbf{G}_{\ell\ell}$ when the weights $\mathbf{w}_\ell$ are rescaled by a factor γ. The previous layer's activations, and hence $\mathbf{A}_{\ell-1}$, are unchanged. However, the next layer's pre-activations $\mathbf{s}_\ell$ scale proportionally to $\mathbf{w}_\ell$, and hence (following the reasoning of Section 4.3) the pseudogradient $\mathcal{D}\mathbf{w}_\ell$ scales as γ^{-1}, and the second factor $\mathbf{S}_\ell$ scales as γ^{-2}. Hence, $\mathbf{G}_{\ell\ell}$ scales as γ^{-2}.

Without WD regularization, the norm of the weights increases over the course of training, according to the argument from Section 4.3. As the weights get larger, $\mathbf{G}_{\ell\ell}$ gets smaller, and the damping term $\eta\mathbf{I}$ comes to dominate the preconditioner. I.e., due to the growing weight norm, K-FAC behaves increasingly like a first-order optimizer. WD keeps the weight norm from growing too large, thereby allowing K-FAC to retain more of its second-order behavior.

5.3 Mechanism 3: Jacobian Norm Regularization

The first two mechanisms both involved the effects of WD on the training dynamics, rather than what we traditionally think of as regularization. Does it ever acts as a traditional regularizer? The answer is yes, but in a pretty surprising way.

If we apply WD when using an optimizer that preconditions the gradient by a matrix $\mathbf{C}^{-1}$, then we can reformulate the update as the preconditioned update on a modified cost function:

It only makes sense to talk about the $\mathbf{C}$-norm for architectures which are not invariant to rescaling. In a scaling-invariant architecture, one could make the $\mathbf{C}$-norm arbitrarily small by rescaling the weights, so the regularizer doesn't constrain the model's capacity. Hence, in this section we assume there is no BN.

$$\begin{aligned}\mathbf{w}^{(k+1)} &\leftarrow (1-\alpha\lambda)\mathbf{w}^{(k)} - \alpha\mathbf{C}^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)}) \\ &= \mathbf{w}^{(k)} - \alpha\mathbf{C}^{-1}(\lambda\mathbf{C}\mathbf{w}^{(k)} + \nabla\mathcal{J}(\mathbf{w}^{(k)})) \\ &= \mathbf{w}^{(k)} - \alpha\mathbf{C}^{-1}\nabla[\mathcal{J}(\mathbf{w}^{(k)}) + \tfrac{\lambda}{2}\|\mathbf{w}^{(k)}\|_{\mathbf{C}}^2],\end{aligned} \tag{27}$$

where $\|\cdot\|_{\mathbf{C}}$ denotes the **C-norm**

$$\|\mathbf{v}\|_{\mathbf{C}} = \sqrt{\mathbf{v}^\top\mathbf{C}\mathbf{v}}.$$

As a sanity check, SGD corresponds to the choice $\mathbf{C} = \mathbf{I}$, and therefore the $\mathbf{C}$-norm is simply the standard L^2 norm that WD is known to regularize.

For other optimizers, it's not always easy to assign the $\mathbf{C}$-norm an intuitive interpretation. However, when Gauss-Newton optimization is applied to a multilayer perceptron with ReLU activations and no BN, it can be shown that the norm regularized by WD is proportional to the Frobenius norm of the input-output Jacobian $\mathbf{J}_{\mathbf{zx}}$, i.e. the Jacobian of the network's outputs with respect to its inputs. A smaller Jacobian norm corresponds to a smoother (more slowly varying) function. Interestingly, this is a way of constraining the model's capacity *in function space*, as opposed to the weight space regularization typically associated with WD.

If the activations are linear rather than ReLU, then the K-FAC Gauss-Newton norm is also proportional to the Jacobian norm.

The equivalence with Jacobian norm regularization only holds exactly in a fairly restricted case. However, it appears empirically that K-FAC combined with WD leads to smaller input-output Jacobian norms for a wider variety of architectures in practice. The details can be found in Zhang et al. (2019).

6 Discussion

The topics of this lecture (especially BN) have been the source of much hand-wringing about how we don't understand neural nets. But, much like a Scooby Doo episode, once you pull the cloth off the supposedly paranormal phenomenon, it is almost always revealed to have a mundane explanation. *Deep learning is not deeply mysterious.*

Unfortunately, our field doesn't do a very good job of *conveying to practitioners* what we understand. In writing this lecture, it was hard for me to find papers or tutorials that explained the ideas clearly. Part of the problem is likely a publication bias: unfortunately, it's easier to get papers accepted if they use fancy and difficult techniques from math and physics. Conversely, the better a job one does of explaining a phenomenon, the less impressive the contribution appears. Many of the ideas presented in this lecture rely only on basic arithmetic, and would likely be hard to publish in one of the major conferences. And once the papers are published, they tend not to be very discoverable by practitioners (witness, e.g., the rather low citation counts for many papers cited in this chapter which contain ideas a lot of deep learning practitioners would benefit from knowing).

Here's another downer: seemingly fundamental differences between algorithms often turn out to be reducible to mundane issues such as hyperparameter tuning. E.g., in our field, there is widely believe to be a generalization gap between SGD and Adam, supposedly resulting from the tendency of adaptive gradient methods to coverge to "sharp" rather than "flat" minima. When we tried to chase down this effect in a few cases where we'd observed it, it went away once we carefully tuned learning rate schedules for both algorithms. Choi et al. (2019) performed a thorough, systematic comparison of SGD, RMSprop, and Adam, and found that they behave in the ways we'd intuitively expect, but only if one does a particularly careful hyperparameter sweep for each method.

The idea of sharp and flat minima will be discussed in Chapter 8.

Even though the algorithmic techniques in this chapter are not deeply mysterious in the ways that are often claimed, they do present a lot of practical difficulties when it comes to engineering and understanding a particular deep learning system. The reason is that many standard neural net components (such as the ones discussed in this chapter) introduce confounds whereby one change to the model influences seemingly unrelated aspects of the training dynamics. For instance,

- For Adam, RMSprop, etc., larger batch sizes result in more gradient noise, and hence smaller steps.
- With BN, the batch size affects the amount of stochastic regularization.
- With homogeneous normalizers such as BN, WN, and LN, the norm of the weights affects the effective learning rate.
- Different optimizers (SGD, Adam, K-FAC, etc.) have different effective learning rate schedules when combined with homogeneous normalizers.
- For many architectures, weight decay fundamentally affects the training dynamics, so it can't be tuned independently of the optimizer (as we might expect for a regularization hyperparameter).

Effects such as these result in a lot of counterintuitive behaviors and make it hard to properly tune hyperparameters. I hope that, as more researchers understand the effects described in this chapter, the field will converge to a set of best practices that reduce the number of gotchas when tuning a deep learning system.

References

Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, and Cyril Zhang. Disentangling adaptive gradient methods from learning rates. arXiv:2002.11803, 2020.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. In *Neural Information Processing Systems*, 2016.

Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*. Springer, 2012.

Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. On empirical comparisons of optimizers for deep learning. arXiv:1910.05446, 2019.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. An investigation into neural net optimization via Hessian eigenvalue density. In *International Conference on Machine Learning*, 2019.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. arXiv:1802.09568, 2018.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Neural Information Processing Systems*, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, 2015.

Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems*, 2012.

Frederik Kunstner, Lukas Balles, and Philipp Hennig. Limitations of the empirical Fisher approximation for natural gradient descent. In *Neural Information Processing Systems*, 2019.

Yann LeCun, Ido Kanter, and Sara A Solla. Second order properties of error surfaces: Learning time and generalization. In *Neural Information Processing Systems*, 1991.

Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. In *International Conference on Learning Representations*, 2020.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

Kevin Luk and Roger Grosse. A coordinate-free construction of scalable natural gradient. arXiv:1808.10340, 2018.

J. Martens and R. Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning*, 2015.

Yann Ollivier. Riemannian metrics for neural networks I: feedforward networks. *Information and Inference*, 4(2):108–153, 2015.

Vardan Papyan. Traces of class/cross-class structure pervade deep learning spectra. arXiv:2008.11865, 2020.

Ali Rahimi. Back when we were young. NeurIPS Test of Time Talk, 2017.

Simon Roburin, Yann de Mont-Marin, Andrei Bursuc, Renaud Marlet, Patrick Perez, and Mathieu Aubry. A spherical analysis of Adam with batch normalization. arXiv:2006.13382, 2020.

Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Neural Information Processing Systems*, 2016.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksandar Madry. How does batch normalization help optimization? In *Neural Information Processing Systems*, 2018.

Tijmen Tieleman and Geoffrey E. Hinton. Lecture 6.5 - RMSProp, COURSERA: Neural Networks for Machine Learning. Technical Report, 2012.

Twan van Laarhoven. L2 regularization versus batch and weight normalization. In *Neural Information Processing Systems*, 2017.

Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three mechanisms of weight decay regularization. In *International Conference on Learning Representations*, 2019.