Chapter 4
Second-Order Optimization

Roger Grosse

1 Introduction

I've been delaying the topic of second-order optimization, because I wanted to introduce the matrices **H**, **G**, and **F** in settings where their usefulness was more readily apparent. Also, I wanted to emphasize that these matrices are broadly useful in many settings other than optimization (and we'll continue to see examples of this throughout the course). But second-order optimization is where these matrices, and approximations thereof, were first pioneered in our field, and remains one of the primary use cases. So let's finally confront second-order optimization.

The term "second-order optimization" carries different meanings to different people. For example, in the field of numerical optimization, there is a fundamental distinction between algorithms with first-order and second-order convergence rates, i.e. whether the number of significant digits in the answer increases linearly or quadratically with the number of iterations. However, this distinction is not so significant in machine learning, where our updates are stochastic, and therefore have worse asymptotics than deterministic optimizers anyway (see Chapter 7). Similarly, natural gradient descent is closely related to Riemannian manifold gradient descent in the field of Riemannian optimization; in that field, it is considered a first-order optimization algorithm, and is explicitly contrasted with Riemannian manifold generalizations of second-order optimizers such as Newton-Raphson. For purposes of this course, *second-order optimization* will simply refer to optimization algorithms that use second-order information, such as the matrices **H**, **G**, and **F**. Hence, stochastic Gauss-Newton optimizers and natural gradient descent will both be considered second-order optimizers.

We start by giving several different perspectives on what second-order optimization is trying to achieve. First, we view the algorithms as iteratively minimizing quadratic approximations to the cost function. We then view them as preconditioners which transform the space to be better conditioned; this can be a more productive way to think about faster and less accurate curvature approximations. We consider invariance to reparameterizations, one of the key properties that motivates natural gradient. Finally, we adopt a proximal optimization perspective, where second-order information is used to prevent the optimizer from forgetting previously learned information after each update. All of these perspectives are useful for understanding different aspects of the algorithms we'll consider.

Computing exact second-order updates is impractical for all but the smallest neural networks, because they require solving linear systems with dimensions in the millions. Therefore, we need some way to approximate the second-order matrices, or to approximately solve the linear system. In the field of deep learning, we have two sets of tools available to us. The first — which we've already been using in this course, and which is foundational to scientific computing — is to use the exact second-order matrices for a batch of data, and to compute with them using MVPs. For instance, we can approximately solve linear systems using CG.

The second approach is to fit parametric approximations to $\mathbf{G}$, most notably the K-FAC approximation. The field of probabilistic graphical models has given us a powerful set of tools we can use to define probabilistic models that exploit the structure of a problem. If we impose the right kinds of structure, we can compactly represent very high-dimensional probability distributions, and efficiently perform operations we need for second-order optimization, such as computing the inverse covariance.

The use of preconditioning matrices, which approximate large matrices with ones that are efficiently invertible, is a foundational technique in numerical computing. However, constructing preconditioners *by fitting statistical models* is, to the best of my knowledge, a novel contribution by the field of deep learning.

2 What are we Trying to Achieve?

We now overview several different motivations for using second-order information in optimization.

2.1 Minimizing Quadratic Approximations

In Chapter 1, we analyzed the dynamics of gradient descent on convex quadratic objectives $\mathbf{w}^\top \mathbf{A} \mathbf{w}$ in great detail. We saw that the maximum step size is determined by the maximum eigenvalue of the matrix $\mathbf{A}$, and that convergence is much slower along low curvature directions. Hence, the asymptotic convergence rate is determined by the condition number of $\mathbf{A}$, i.e. the ratio of the maximum and minimum nonzero eigenvalues. In Chapter 2, we considered the second-order Taylor approximation of a cost function around a point $\mathbf{w}_0$:

$$\mathcal{J}_{\text{quad}}(\mathbf{w}) = \mathcal{J}(\mathbf{w}_0) + \nabla \mathcal{J}(\mathbf{w}_0)^\top (\mathbf{w} - \mathbf{w}_0) + \tfrac{1}{2} (\mathbf{w} - \mathbf{w}_0)^\top \mathbf{H} (\mathbf{w} - \mathbf{w}_0). \quad (1)$$

Hence, close to the stationary point, the gradient descent dynamics are well approximated by those of a quadratic objective. If $\mathbf{w}_\star$ is a local minimum, then $\mathbf{H} \succeq \mathbf{0}$, and the quadratic approximation is convex. Hence, the asymptotic convergence rate is determined by the condition number of $\mathbf{H}$. In the linear autoencoder example, we saw that slow convergence of certain parts of the model can be reflected in the eigenvalues of $\mathbf{H}$ even far from the optimum.

This Taylor approximation motivates a classic optimization algorithm called **Newton-Raphson**, or sometimes just **Newton's Method**. This algorithm is best motivated in the case where $\mathcal{J}$ is convex, and hence $\mathbf{H} \succeq \mathbf{0}$. In this case, any critical point of $\mathcal{J}$ is also a global optimum, so therefore our goal is to find a critical point, i.e. a point $\mathbf{w}_\star$ such that $\nabla \mathcal{J}(\mathbf{w}_\star) = 0$. The first-order Taylor approximation to $\nabla \mathcal{J}$ around the current weights $\mathbf{w}_0$ is given by:

$$\nabla \mathcal{J}(\mathbf{w}) \approx \nabla \mathcal{J}(\mathbf{w}_0) + \mathbf{H}(\mathbf{w} - \mathbf{w}_0). \quad (2)$$

By setting this to zero and solving for $\mathbf{w}$, we arrive at the approximation to the optimality condition:

$$\mathbf{w}' = \mathbf{w}_0 - \mathbf{H}^{-1}\nabla\mathcal{J}(\mathbf{w}_0). \tag{3}$$

This process can then be repeated at the new $\mathbf{w}$, and if all goes well, $\mathbf{w}$ will approach $\mathbf{w}_\star$, and the Taylor approximation to $\nabla\mathcal{J}$ will get increasingly accurate in the vicinity of $\mathbf{w}_\star$.

An alternative viewpoint is that we are repeatedly minimizing second-order Taylor approximations to $\mathcal{J}$ (Eqn. 1). In particular, because $\mathbf{H} \succeq \mathbf{0}$, the optimum of Eqn. 1 is given by $\mathbf{w}' = \mathbf{w}_0 - \mathbf{H}^{-1}\nabla\mathcal{J}(\mathbf{w}_0)$. Because Newton-Raphson is based on a second-order Taylor approximation to the cost function, whereas gradient descent is based on a first-order Taylor approximation (i.e. it only uses the gradient), we should expect Newton-Raphson to converge faster.

This vanilla version of Newton-Raphson is not by itself guaranteed to converge efficiently (or at all), or even to decrease the cost function in each iteration. The problem is that the algorithm might take a large step, thereby moving far enough from $\mathbf{w}_0$ that the second-order Taylor approximation is no longer accurate. Consider an example of this which is particularly relevant to machine learning: the softmax-cross-entropy loss for a binary classifier, for a positive training example, as a function of the logit z. The loss, and its first and second derivatives, are given by:

Normally we are optimizing with respect to the weights, not the logits, but this problem still occurs.

$$\mathcal{J}(z) = \log(1+\exp(-z)) \qquad \mathcal{J}'(z) = -\sigma(-z) \qquad \mathcal{J}''(z) = \sigma(z)\sigma(-z), \tag{4}$$

where $\sigma(z) = 1/(1+\exp(-z))$ is the logistic function. For $z \ll 0$ (i.e. a very incorrect prediction), the cost is very close to linear: $\mathcal{J}'(z) \approx -1$, and $\mathcal{J}''(z) \approx \exp(z)$. Hence, Newton-Raphson takes a very big step:

$$z \leftarrow z_0 - \frac{\mathcal{J}'(z)}{\mathcal{J}''(z)} \approx z_0 + \exp(-z).$$

To fix this problem, we need to somehow **dampen** the update by preventing it from moving too far in low-curvature directions. One way to do this is to add a proximity term to Eqn. 1 penalizing the Euclidean distance from the previous iterate:

$$\begin{aligned}\mathbf{w}^{(k+1)} &= \arg\min_{\mathbf{w}} \mathcal{J}_{\text{quad}}(\mathbf{w}) + \tfrac{\eta}{2}\|\mathbf{w} - \mathbf{w}^{(k)}\|^2 \\ &= \arg\min_{\mathbf{w}} \nabla\mathcal{J}(\mathbf{w}^{(k)})^\top\mathbf{w} + \tfrac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top(\mathbf{H} + \eta\mathbf{I})(\mathbf{w} - \mathbf{w}^{(k)}) \\ &= \mathbf{w}^{(k)} - (\mathbf{H} + \eta\mathbf{I})^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)}),\end{aligned} \tag{5}$$

where $\eta > 0$ is the **damping parameter**. Hence, the damped Newton-Raphson update is just like the vanilla Newton-Raphson update, except that $\mathbf{H}$ is replaced by $\mathbf{H} + \eta\mathbf{I}$. To understand the effect of this difference, observe that $\mathbf{H}^{-1}$ and $(\mathbf{H} + \eta\mathbf{I})^{-1}$ are both codiagonalizable with $\mathbf{H}$, i.e. they share the same eigenvectors. If the eigenvalues of $\mathbf{H}$ are denoted as ν_i, then the corresponding eigenvalues of $\mathbf{H}^{-1}$ are given by ν_i^{-1}, and the corresponding eigenvalues of $(\mathbf{H} + \eta\mathbf{I})^{-1}$ are given by $(\nu_i + \eta)^{-1}$. For $\nu_i \gg \eta$ (i.e. high curvature directions), these two values are nearly the same, i.e. both the undamped and damped updates shrink the step by similar amounts in that direction. For $\nu_i \ll \eta$ (i.e. low curvature directions), $(\nu_i + \eta)^{-1} \approx \eta^{-1}$. Therefore, while the undamped algorithm is willing to stretch the update by an arbitrarily large factor, the damped update is willing to stretch it by at most a factor of η^{-1}. Both algorithms behave similarly in high curvature directions, while the damped version makes more conservative updates in low curvature directions.

The above discussion all assumes that $\mathcal{J}$ is convex. Applying the vanilla Newton-Raphson update to non-convex objectives makes little sense, because it searches for critical points (which include saddle points), rather than local optima. In Chapter 2, we observed that unless you get really unlucky, gradient descent escapes saddle points. So in this sense, Newton-Raphson behaves *worse* that gradient descent for non-convex problems.

In deep learning, the standard solution to this problem is to replace $\mathbf{H}$ with the Gauss-Newton Hessian $\mathbf{G} = \mathbb{E}[\mathbf{J}_{\mathbf{zw}}^\top \mathbf{H}_{\mathbf{z}} \mathbf{J}_{\mathbf{zw}}]$ (see Chapter 2), giving an update rule called the **Gauss-Newton algorithm**. We observed in Chapter 2 that as long as $\mathbf{H}_{\mathbf{z}}$ is PSD (which can be guaranteed by choosing a convex loss function), $\mathbf{G}$ is PSD as well. The damped matrix $\mathbf{G} + \eta\mathbf{I}$, therefore, is positive definite. Since positive definite matrices are closed under inverses, this implies $(\mathbf{G} + \eta\mathbf{I})^{-1}$ is also positive definite.

Sometimes, the name *Gauss-Newton algorithm* is reserved for the case of squared error loss, and the more general version is referred to as the *generalized Gauss-Newton algorithm.* We'll simply use *Gauss-Newton* for the general case.

In general, preconditioning the gradient descent update with any positive definite matrix results in a **descent direction**. By this, I mean that it has a negative dot product with the gradient, and therefore is guaranteed to reduce the cost for a small enough step size. To see, this, let $\mathbf{v} = -\mathbf{A}\nabla\mathcal{J}(\mathbf{w}^{(k)})$ for some positive definite matrix $\mathbf{A}$. Assume we are not at a critical point, so that $\nabla\mathcal{J}(\mathbf{w}^{(k)}) \neq \mathbf{0}$. The dot product with the gradient is given by:

$$\nabla\mathcal{J}(\mathbf{w}^{(k)})^\top \mathbf{v} = -\nabla\mathcal{J}(\mathbf{w}^{(k)})^\top \mathbf{A} \nabla\mathcal{J}(\mathbf{w}^{(k)}) < 0, \tag{6}$$

where the inequality follows from the definition of a positive definite matrix. Since $(\mathbf{G} + \eta\mathbf{I})^{-1}$ is positive semidefinite, this implies the Gauss-Newton algorithm gives a descent direction.

This same argument applies to natural gradient descent. Any pullback metric, including the Fisher information matrix, is positive semidefinite. Hence, the same argument shows that a damped natural gradient update gives a descent direction.

A major obstacle to computing the Newton-Raphson and Gauss-Newton updates is that the formula involves the inverse of $\mathbf{H}$ or $\mathbf{G}$. Inverting a matrix is an $\mathcal{O}(D^3)$ operation, where D is the dimension of that matrix. For neural net optimization, D is the number of parameters, which is typically in the millions or sometimes even billions, so inversion is out of the question. Note that we don't literally need to invert the matrix, and in fact numerical algorithms avoid explicit matrix inversion wherever possible, for reasons of numerical stability. However, we do need to solve a linear system $(\mathbf{G} + \eta\mathbf{I})\mathbf{v} = -\nabla\mathcal{J}(\mathbf{w}^{(k)})$, which is also an $\mathcal{O}(D^3)$ operation if one requires an exact solution. So the point remains that exactly computing the Newton-Raphson or Gauss-Newton update is impractical for modern networks.

Therefore, the Newton-Raphson and Gauss-Newton algorithms are best viewed as idealized algorithms which we aim to approximate. This has, of course, been a fundamental topic in numerical optimization for many decades, and there are numerous approximations. One such approximation is to approximately solve the linear system using conjugate gradient. This is the basis of **CG-Newton** optimizers, and we'll discuss this in more detail in

Section 4. Relatedly, rather than explicitly solving the quadratic, one can use a nonlinear CG algorithm. **Quasi-Newton methods** are a class of algorithms that efficiently exploit second order information using only first derivatives; the most notable examples are **BFGS** and **L-BFGS**. These approaches have been remarkably successful at optimizing a deterministic objective function, and are standard off-the-shelf tools in numerical libraries such as SciPy. These techniques are covered in standard texts such as the excellent Nocedal and Wright (2006) and Bertsekas (2016).

2.2 Preconditioning

Rather than minimizing a quadratic approximation to $\mathcal{J}$, a more modest goal is to transform the problem so that it is **well-conditioned**, which for our purposes is an informal term meaning that the differences in curvature between different directions are not too extreme.

Often, one defines "well-conditioned" this in terms of the **condition number** of $\mathbf{G}$ or $\mathbf{H}$. However, modern neural nets are often overparameterized, which implies that some of the eigenvalues are 0. The optimization problem is nonconvex, so some eigenvalues may be negative. Furthermore, even some of the positive eigenvalues correspond to directions which are unimportant, or even undesirable, to optimize in, e.g. because they correspond to overfitting. Intuitively speaking, we'd like to avoid the situation where important directions have very low curvature, but so far we haven't found a reliable way to quantify this desideratum.

In Chapter 1, we observe that gradient descent is *not* invariant to linear reparameterizations of the optimization variables. In particular, consider the reparameterization

$$\mathbf{w} = \mathcal{T}(\tilde{\mathbf{w}}) = \mathbf{R}\tilde{\mathbf{w}} + \mathbf{b}, \tag{7}$$

where $\mathbf{R}$ is an invertible square matrix (not necessarily symmetric), and $\mathbf{b}$ is a vector. The inverse transformation is given by:

$$\tilde{\mathbf{w}} = \mathbf{R}^{-1}(\mathbf{w} - \mathbf{b}).$$

Similarly to Chapter 1, we can write the cost function in terms of $\tilde{\mathbf{w}}$:

$$\tilde{\mathcal{J}}(\tilde{\mathbf{w}}) = \mathcal{J}(\mathcal{T}(\tilde{\mathbf{w}})).$$

By the Chain Rule, the gradient in the transformed space is given by:

$$\nabla\tilde{\mathcal{J}}(\tilde{\mathbf{w}}) = \mathbf{R}^\top \nabla\mathcal{J}(\mathcal{T}^{-1}(\tilde{\mathbf{w}})).$$

So suppose we carry out gradient descent in the transformed space. I.e., we transform our current iterate, compute the gradient descent update on $\tilde{\mathcal{J}}$, and transform the iterate back to the original space:

$$\begin{aligned}\mathbf{w}^{(k+1)} &= \mathcal{T}(\tilde{\mathbf{w}}^{(k)} - \alpha\nabla\tilde{\mathcal{J}}(\tilde{\mathbf{w}}^{(k)})) \\ &= \mathcal{T}(\tilde{\mathbf{w}}^{(k)} - \alpha\mathbf{R}^\top\nabla\mathcal{J}(\mathbf{w}^{(k)})) \\ &= \mathbf{w}^{(k)} - \alpha\mathbf{R}\mathbf{R}^\top\nabla\mathcal{J}(\mathbf{w}^{(k)}).\end{aligned} \tag{8}$$

Therefore, doing gradient descent in the transformed space is equivalent to doing gradient descent in the original space, but preconditioned by $\mathbf{R}\mathbf{R}^\top$. The matrix $\mathbf{R}\mathbf{R}^\top$ is called the **preconditioner**. Note that we never have to construct $\mathbf{R}$ or compute $\tilde{\mathbf{w}}$ explicitly; rather, the transformed space is entirely implicit, and all we need algorithmically is to compute MVPs with the preconditioner.

The offset term $\mathbf{b}$ doesn't matter, consistent with our observation in Chapter 1 that gradient descent is invariant to rigid transformations, including translation.

Conversely, preconditioning the gradient descent update by a PSD matrix $\mathbf{A}$ is equivalent to gradient descent in a transformed space, where the transformation matrix $\mathbf{R}$ is such that $\mathbf{R}\mathbf{R}^\top = \mathbf{A}$. For instance, we can use the **matrix square root**, defined by $\mathbf{R} = \mathbf{A}^{1/2} = \mathbf{Q}\mathbf{D}^{1/2}\mathbf{Q}^\top$, where

$\mathbf{A} = \mathbf{Q}\mathbf{D}\mathbf{Q}^\top$ is the spectral decomposition of $\mathbf{A}$. Hence, the damped Gauss-Newton update is equivalent to doing gradient descent in a space that's stretched out by a factor of $(\mathbf{G} + \eta\mathbf{I})^{1/2}$.

Consider a strictly convex problem, so that $\mathbf{H} \succ \mathbf{0}$. It can be shown that the Hessian in the transformed space is given by:

$$\tilde{\mathbf{H}} = \nabla^2 \tilde{\mathcal{J}}(\tilde{\mathbf{w}}) = \mathbf{R}^\top \mathbf{H} \mathbf{R}. \tag{9}$$

Hence, if $\tilde{\mathbf{H}}$ is much better conditioned than $\mathbf{H}$, we'd expect the preconditioned gradient descent update to converge more efficiently than ordinary gradient descent. As an extreme case, Newton-Raphson preconditions by $\mathbf{H}^{-1}$, implying that $\mathbf{R} = \mathbf{H}^{-1/2}$, and

$$\tilde{\mathbf{H}} = \mathbf{H}^{-1/2}\mathbf{H}\mathbf{H}^{-1/2} = \mathbf{I}.$$

I.e., Newton-Raphson can be seen as preconditioning such that the curvature becomes isotropic (spherical). Similarly, the damped Newton-Raphson update gives

$$\tilde{\mathbf{H}} = (\mathbf{H} + \eta\mathbf{I})^{-1/2}\mathbf{H}(\mathbf{H} + \eta\mathbf{I})^{-1/2}.$$

You can check that $\tilde{\mathbf{H}}$ is codiagonalizable with $\mathbf{H}$, and that it has eigenvalues of approximately 1 for $\nu_i \gg \eta$ and approximately $\eta^{-1}\nu_i$ for $\nu_i \ll \eta$. Hence, the damped upate converges at roughly the same rate for all curvatures above η, and more slowly for curvatures below η.

The reason that preconditioning is useful is that often even a very crude approximation to $\mathbf{H}^{-1}$ can substantially improve the conditioning of the optimization problem. A classic choice from the field of optimization is to use a diagonal approximation to $\mathbf{H}$. Inversion of a diagonal matrix simply requires taking the inverses of the diagonal entries, so this is an $\mathcal{O}(D)$ operation (in contrast to $\mathcal{O}(D^3)$ for inverting $\mathbf{H}$). One can often develop much better preconditioners by exploiting the structure of an optimization problem; the incomplete Cholesky factorization is a classic example from the field of numerical optimization.

Previous lectures have highlighted connections between the Hessian $\mathbf{H}$, the Gauss-Newton Hessian $\mathbf{G}$, the classical Gauss-Newton matrix, pullback metrics (also denoted by $\mathbf{G}$), and the Fisher information matrix $\mathbf{F}$. Since even crude approximations to $\mathbf{H}$ can be very useful for preconditioning, we can design preconditioners by approximating any one of these matrices in place of $\mathbf{H}$. In Section 5, we'll see a particularly useful class of preconditioners for neural nets, based on Kronecker-factored approximations to $\mathbf{G}$ or $\mathbf{F}$.

So far, our discussion of preconditioners has focused on gradient descent. However, preconditioning is a much more broadly useful tool. For instance, we saw in Chapter 2 that we can approximately solve linear systems involving $\mathbf{H}$, $\mathbf{G}$, etc. using MVPs and conjugate gradient (CG). Analogously to gradient descent, the convergence of CG depends on the condition number, so preconditioning can substantially speed up convergence. Just like preconditioned gradient descent can be formulated without ever constructing $\mathbf{R}$ explicitly, the same is true of preconditioned CG and other similar methods. Library routines implementing CG typically take as an optional argument a function computing an MVP with the preconditioner.

2.3 Invariance to Reparameterization

Invariance of an optimization algorithm to transformations of the parameter space has been a running theme so far in the course. In Chapter 1, as well as the preceding section, we observed that gradient descent is invariant to rigid transformations of the parameter space (i.e., rotations, reflections, and translations), but is not invariant to more general linear transformations. In Chapter 1, we saw that the arbitrary choice of units can have a significant effect on the convergence of gradient descent for linear regression, and that it's therefore advantageous to normalize the inputs to zero mean and unit variance. We noted that this solution works only for shallow models like linear regression; even with normalized inputs, deep neural nets still suffer from these optimization issues due to the problem of internal covariate shift.

When the ill-conditioning can't be solved using explicit normalization, we can instead design an optimizer to be invariant to certain transformations which we believe ought not influence the optimization trajectory. In Chapter 3, we considered proximal optimization methods, and saw that invariance could be achieved by choosing a proximity term (such as KL divergence) defined in terms of the function itself rather than the parameters. Since natural gradient is based on a second-order Taylor approximation to proximal optimization, it inherits the invariance properties, up to the first order. We briefly saw some mathematical tools that can be used to construct various mathematical objects in a coordinate-free way. Algorithms designed from such building blocks achieve parameterization invariance for free.

It is also possible to show that the (undamped) Newton-Raphson and Gauss-Newton updates are invariant to affine transformations of the parameter space. Consider an affine transformation as given by Eqn. 7. Like in Chapter 1, we can make an inductive argument where we assume $\mathbf{w}^{(k)} = \mathcal{T}(\tilde{\mathbf{w}}^{(k)})$ and show the same for step $k = 1$. It can be shown using the Chain Rule that the Hessian in the transformed space is given by:

$$\tilde{\mathbf{H}} = \nabla^2 \tilde{\mathcal{J}}(\tilde{\mathbf{w}}) = \mathbf{R}^\top [\nabla^2 \mathcal{J}(\mathcal{T}(\mathbf{w}))] \mathbf{R} = \mathbf{R}^\top \mathbf{H} \mathbf{R}.$$

Combined with the formula $\nabla \tilde{\mathcal{J}}(\tilde{\mathbf{w}}) = \mathbf{R}^\top \nabla \mathcal{J}(\mathcal{T}(\tilde{\mathbf{w}}))$ (Eqn. 8), this gives us the Newton update to $\tilde{\mathbf{w}}$:

$$\begin{aligned}
\tilde{\mathbf{w}}^{(k+1)} &= \tilde{\mathbf{w}}^{(k)} - \alpha \tilde{\mathbf{H}}^{-1} \nabla \tilde{\mathcal{J}}(\tilde{\mathbf{w}}) \\
&= \tilde{\mathbf{w}}^{(k)} - \alpha [\mathbf{R}^\top \mathbf{H} \mathbf{R}]^{-1} \mathbf{R}^\top \nabla \mathcal{J}(\mathcal{T}(\tilde{\mathbf{w}}^{(k)})) \\
&= \tilde{\mathbf{w}}^{(k)} - \alpha \mathbf{R}^{-1} \mathbf{H}^{-1} \nabla \mathcal{J}(\mathcal{T}(\tilde{\mathbf{w}}^{(k)})) \\
&= \mathcal{T}^{-1}(\mathbf{w}^{(k)} - \alpha \mathbf{H}^{-1} \nabla \mathcal{J}(\mathbf{w}^{(k)})) \\
&= \mathcal{T}^{-1}(\mathbf{w}^{(k+1)}).
\end{aligned}$$

Therefore, by induction, $\mathbf{w}^{(k)} = \mathcal{T}(\tilde{\mathbf{w}}^{(k)})$ for all k. I.e., Newton-Raphson is invariant to affine reparameterizations. The exact same derivation holds with $\mathbf{G}$ in place of $\mathbf{H}$, so Gauss-Newton is invariant to affine reparameterizations as well.

Note that invariance only holds for the *undamped* versions of these algorithms. Damping penalizes Euclidean distance in parameter space, so it is inherently tied to the parameterization. Because the damped algorithms behave similarly to the undamped algorithms in the high curvature directions (see previous section), the damped algorithms still achieve partial invariance. Note that full invariance might not even be desirable: we already noted in Chapter 1 that gradient descent's emphasis on high curvature directions can sometimes be a useful inductive bias. We'll see more examples of this later in the course.

Invariance is a useful design principle for optimization algorithms, since we can design efficient optimizers that satisfy a more restricted set of invariance properties. For instance, if we are interested in compensating for internal covariate shift, we can design an algorithm to be invariant to affine transformations of the activations in each layer. This point will be discussed further in Chapter 5 in the context of batch norm.

2.4 Proximal Optimization

In Chapter 3, we motivated proximal optimization, and its approximations such as natural gradient descent, in terms of doing "gradient descent in output space." This provides a useful intuition for neural net training, but there are two key differences: (1) when we train a neural net (or most other machine learning models), we update the parameters on one batch at a time, and (2) we would like the model to generalize to new data. If we literally did gradient descent on the outputs, we would have to iterate through the entire training set in order to fit it (and hence convergence could even be slower than for SGD), and there is no guarantee we'd generalize to new data.

Roughly speaking, we can think of second-order optimization algorithms for neural nets as choosing updates to minimize some combination of the following three factors:

Loss on the current batch. This term is conceptually straightforward: we'd like to improve the predictions on the current batch.

Function space proximity (FSP). We'd like to change the network's predictions as little as possible, on average. This prevents the current update from screwing up the predictions on examples we've visited in the past.

Weight space proximity (WSP). We'd like to change the weights as little as possible. One reason to do this is to ensure that any second-order approximations to the loss or to FSP remain accurate, as in the damped Newton-Raphson and Gauss-Newton algorithms. Another reason is that penalizing weight space distance seems to regularize the algorithm to produce smoother functions, for reasons we're just beginning to understand. (We'll say more about this effect in Chapter 6.)

Because of the relationships between the Hessian and pullback metrics (see Chapters 2 and 3), there isn't always a clean separation between the loss and FSP terms. Some algorithms, such as Hessian-free optimization, can be interpreted in multiple ways (see Section 4).

All of these factors can be summarized into an idealized proximal optimization algorithm which trades off all three factors. Letting $\mathcal{B}^{(k+1)}$ denote the

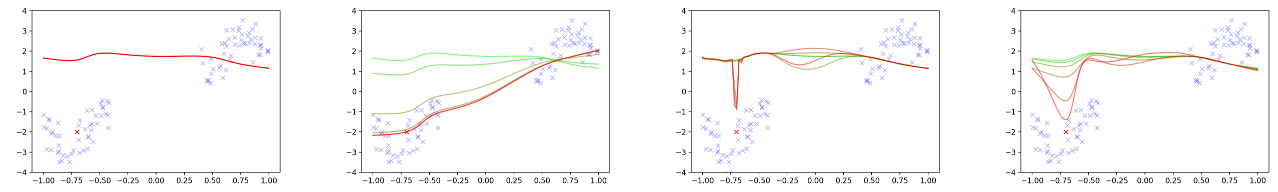

Figure 1: Example of idealized proximal optimization applied to a toy regression problem with a batch size of 1 (shown in red). From left to right: **(1)** the current model, **(2)** only WSP, **(3)** only FSP, and **(4)** both proximity terms. The weight of the proximity term(s) is decreased going from **green** to **red**.

current batch, we have:

$$\mathbf{w}^{(k+1)} \leftarrow \arg\min_{\mathbf{w}} \frac{1}{|\mathcal{B}^{(k+1)}|} \sum_{i \in \mathcal{B}^{(k+1)}} \mathcal{L}(f(\mathbf{x}^{(i)}, \mathbf{w}), \mathbf{t}^{(i)}) + \lambda_{\mathrm{FSP}} \mathbb{E}_{\mathbf{x}}[\rho(f(\mathbf{x}, \mathbf{w}), f(\mathbf{x}, \mathbf{w}^{(k)}))] + \frac{\lambda_{\mathrm{WSP}}}{2} \|\mathbf{w} - \mathbf{w}^{(k)}\|^2. \tag{10}$$

This idealized update is, of course, very expensive to compute, for multiple reasons. First of all, the first term is the cost function itself, so we shouldn't expect the proximal objective to be much easier to minimize than the main optimization problem. Also, the FSP term is difficult for two reasons: (1) it is highly nonlinear, and (2) it is defined in terms of the expectation under the data generating distribution, which we don't have access to.

However, many algorithms can be seen as ways of approximately minimizing Eqn. 10. For instance, SGD uses a first-order Taylor approximation to the loss term, and penalizes WSP but not FSP (see Chapter 3). If we use a second-order Taylor approximation to the loss term, and also penalize WSP, this gives a damped Newton algorithm where both the gradient and the Hessian are estimated from the current batch. If we use a first-order approximation to the loss, a second-order approximation to FSP, and also penalize WSP, we get a damped natural gradient descent algorithm. (The latter two claims will be justified in more detail below.)

It is interesting to optimize Eqn. 10 directly on a toy regression example to understand the impact of both proximity terms, as illustrated in Figure 1. The leftmost figure shows the predictions made by the current model. We compute the update on a batch of size 1; the training example is shown in red. The proximal objective is optimized using BFGS, and the FSP term is approximated using the empirical distribution (i.e. the training data). If we penalize only WSP (i.e. $\lambda_{\mathrm{FSP}} = 0$), then the optimizer makes a global update to the function. (Since SGD penalizes WSP but not FSP, this example gives some intuition into the behavior of SGD.) If we penalize only FSP, it carves a spike around the training example, keeping the rest of the predictions more or less constant (except in the region between the two clusters, where FSP does not matter because the data density is 0). If we penalize both WSP and FSP, then it makes a smoother adjustment. Compared to pure FSP, it is likely to generalize better, but it still enjoys a certain locality that the pure WSP update does not.

3 Iteratively Minimizing the Proximal Objective

As discussed above, the idealized proximal objective (Eqn. 10) is generally hard to minimize exactly. Before we turn to second-order approximations, it's first worth considering whether we can just approximately minimize it using gradient descent. Note that we still need to make one approximation: because the FSP term is defined with respect to the data generating distribution, we need to approximate it with the empirical distribution on a batch of examples. So assume we have two batches $\mathcal{B}$ and $\mathcal{B}'$ which we use for the loss and FSP terms (these batches may be identical, but this isn't required). We compute our update by doing gradient descent on the following cost function:

$$\mathcal{Q}(\mathbf{w}) = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathcal{L}(f(\mathbf{x}^{(i)}, \mathbf{w}), \mathbf{t}^{(i)}) + \frac{\lambda_{\mathrm{FSP}}}{|\mathcal{B}'|}\sum_{i\in\mathcal{B}'} \rho(f(\mathbf{x}^{(i)}, \mathbf{w}), f(\mathbf{x}^{(i)}, \mathbf{w}^{(k)})) + \frac{\lambda_{\mathrm{WSP}}}{2}\|\mathbf{w}-\mathbf{w}^{(k)}\|^2. \tag{11}$$

Now consider the computational cost. Just like in Chapter 2, we'll measure the computational cost in terms of *passes* (either forward or backward passes), since this usually predicts the computational cost of an algorithm to within about a factor of 2. Computing $\nabla\mathcal{Q}(\mathbf{w})$ requires computing gradients of each of the three terms. Computing the loss gradient is just ordinary backprop, so it requires two passes (the forward pass and the backward pass). Computing the FSP gradient requires three passes: a forward pass for $\mathbf{w}^{(k)}$ (which only needs to be done once for each proximal update), and a forward and a backward pass for $\mathbf{w}$. The computational cost of the WSP gradient is negligible. Hence, the cost of computing K gradient updates is $2K$ passes on $\mathcal{B}$ and $2K+1$ passes on $\mathcal{B}'$. If the batch is shared between the loss and FSP terms (i.e. $\mathcal{B} = \mathcal{B}'$), then the forward and backward passes can also be shared, and so the total cost is $2K+1$ passes.

This analysis can be improved slightly. Since $\mathbf{w} = \mathbf{w}^{(k)}$ in the first inner iteration, the extra forward pass is redundant, so the total number of passes required is really $2K$, not $2K+1$.

Whether or not iteratively minimizing Eqn. 11 is useful depends on how important computation time is relative to other factors. In the setting of supervised learning, the dataset is fixed, and the main computational bottleneck is the forward and backward passes through the network. Hence, the total cost of an optimizer can be summarized in terms of the number of passes. Gradient descent requires 2 passes per iteration (the forward and backward passes of backprop). Therefore, in the time required to compute K gradient descent steps on the proximal objective (i.e. $2K+1$ passes), we could instead compute gradient descent updates on K different batches of data. All else being equal, we'd rather compute gradient descent updates on fresh training examples than ones we've recently visited, so iteratively minimizing the proximal objective appears to be *strictly worse* than ordinary SGD.

In cases where disk bandwidth is a bottleneck, it may be advantageous to take multiple gradient steps on the same batch of data.

However, iteratively minimizing Eqn. 11 can pay off in situations bottlenecked by factors other than the computational cost of gradients. The most well-known example is reinforcement learning, where interaction with the environment can be expensive, for instance if it requires actions by a physical robot. Hence, in RL algorithms, we are typically more concerned with **sample efficiency**, i.e. the amount of interaction with the environment, rather than computational efficiency. Therefore, we may be willing to spend a significant amount of computation to get a better update.

One of the state-of-the-art RL algorithms, **Proximal policy optimization (PPO)** (Schulman et al., 2017), does gradient descent on a proximal objective very similar to Eqn. 11. Computational cost is sometimes a concern in RL, however, and there are other well-known RL algorithms which further approximate the proximal optimization using techniques covered later in this lecture: in particular, conjugate gradient (TRPO) (Schulman et al., 2015) and Kronecker-factored approximations (ACKTR) (Wu et al., 2017).

PPO was used in OpenAI's famous Dota2 agent, OpenAI Five.

4 Hessian-Free Optimization

We just saw one way to approximately solve the proximal objective, namely to approximate the cost and proximity terms using a single batch, and apply gradient descent. Can we do better than this by taking a second-order approximation?

The second-order Taylor approximations to the loss and proximity terms around $\mathbf{w}_0$ for a single batch are as follows:

It is common to use ℓ_2 regularization. The Hessian of the ℓ_2 term is the identity matrix, so it has a similar effect to the WSP term.

$$\begin{aligned}
\mathcal{J}_{\text{loss}}(\mathbf{w}) &\approx \mathcal{J}(\mathbf{w}_0) + \nabla\mathcal{J}(\mathbf{w}_0)^\top(\mathbf{w}-\mathbf{w}_0) + \tfrac{1}{2}(\mathbf{w}-\mathbf{w}_0)^\top\mathbf{H}(\mathbf{w}-\mathbf{w}_0) \\
&= \nabla\mathcal{J}(\mathbf{w}_0)^\top\mathbf{w} + \tfrac{1}{2}(\mathbf{w}-\mathbf{w}_0)^\top\mathbf{H}(\mathbf{w}-\mathbf{w}_0) + \text{const} \\
\mathcal{J}_{\text{FSP}}(\mathbf{w}) &\approx \frac{\lambda_{\text{FSP}}}{2}(\mathbf{w}-\mathbf{w}_0)^\top\mathbf{G}(\mathbf{w}-\mathbf{w}_0).
\end{aligned}$$

The WSP term is already quadratic, and so doesn't require approximation. If we intend to minimize the quadratic approximation, we require the quadratic to be convex, so that it have a minimum. Since $\mathbf{H}$ may be indefinite, we approximate it with the Gauss-Newton Hessian $\mathbf{G}$, which is always PSD (assuming a convex loss function). Recall that the pullback metric is equivalent to the GN Hessian when the Bregman divergence is used as the output metric. Therefore, the FSP term is redundant, as it is approximated using the same quadratic form as the loss term. We will drop the FSP term, and simply trade off the loss and WSP. The quadratic objective, up to a constant, is given by:

$$\mathcal{Q}_{\text{quad}}(\mathbf{w}) = \nabla\mathcal{J}(\mathbf{w}_0)^\top\mathbf{w} + \tfrac{1}{2}(\mathbf{w}-\mathbf{w}_0)^\top(\mathbf{G}+\lambda\mathbf{I})(\mathbf{w}-\mathbf{w}_0). \tag{12}$$

(Since we no longer need to distinguish λ_{FSP} from λ_{WSP}, I've dropped the WSP subscript to avoid clutter.) The optimum is given by the damped Gauss-Newton update:

$$\mathbf{w}_\star = \mathbf{w} - (\mathbf{G}+\lambda\mathbf{I})^{-1}\nabla\mathcal{J}(\mathbf{w}_0). \tag{13}$$

Unfortunately, this update is impractical to compute exactly, as it requires solving a very large linear system. However, quadratic objectives do have one advantage over non-quadratic ones from an optimization standpoint: we can approximately minimize them using conjugate gradient (CG) instead of gradient descent (see Chapter 2). Recall that CG is an iterative algorithm which requires a single MVP per step, and whose kth iterate is the *optimal* point within the k-dimensional Krylov subspace, i.e. the minimum possible cost achievable with k MVPs and linear combinations. Hence, CG is guaranteed to optimize at least as fast as gradient descent. Theoretical results show that CG converges substantially faster, in particular its iteration complexity grows roughly as $\sqrt{\kappa}$, whereas gradient descent's iteration complexity grows roughly as κ, where κ is the condition number.

Hessian-free optimization (HF) (Martens, 2010) is an approach to neural net optimization which considers one batch of data at a time, and approximately minimizes Eqn. 12 using CG. (It gets its name because it never explicitly represents the Hessian.) Before we turn to practical details of the algorithm, let's first consider the pros and cons relative to iterative minimization of the proximal objective, and to ordinary SGD. Consider the computational cost of HF. The first iteration requires computing $\nabla \mathcal{J}(\mathbf{w}_0)$, which as usual requires 2 passes. Each iteration additionally requires an MVP with $\mathbf{G}$, which also requires 2 passes to compute (see Chapter 2). Therefore, the cost of approximately minimizing Eqn. 12 using K steps of CG is $2K + 2$ passes.

Compared to gradient descent on the proximal objective, HF has approximately the same computational cost per iteration. HF has the advantage that CG minimizes the quadratic approximation faster than gradient descent can minimize the exact proximal objective (or its quadratic approximation, for that matter). The difference can be substantial if the cost function is ill-conditioned. On the other hand, HF has the disadvantage that it's minimizing the quadratic approximation rather than the exact proximal objective. In general, HF should be preferred due to its faster convergence, but we need to take care to stay close enough to $\mathbf{w}_0$ that the quadratic approximation remains accurate; this is achieved by adaptive damping, as described below.

HF also has the disadvantage that it requires implementing MVPs; this can be a serious impediment in frameworks such as TensorFlow, but JAX makes this very easy.

Compared to ordinary SGD, HF benefits from CG's faster rate of convergence. However, it shares the disadvantage of iterative minimization: in the time that it takes to compute K CG steps on a given batch, SGD could have computed updates on $K + 1$ different batches. Therefore, SGD has higher data throughput, which can be a significant advantage. Which of these two factors wins out is complicated, and we discuss this tradeoff in more detail in Chapter 7, which covers stochastic optimization. Roughly speaking, the more ill-conditioned the cost function is, the more beneficial is CG's improved convergence. On the other hand, the more stochastic the gradient estimates are, e.g. because of label noise or the need to use small batches for memory reasons, the more SGD benefits from higher data throughput. When HF was invented in 2010, deep architectures tended to have very poorly conditioned cost functions, and HF could learn much more efficiently than SGD overall. Since then, advances in initialization and architecture design have given us better-conditioned cost functions for the most commonly used architectures, eroding the advantage of HF. However, it remains possible that new applications may require substantially different architectures for which ill-conditioning remains a big problem (see, e.g., Pfau et al. (2019)'s work on neural nets for quantum simulation).

4.1 Adaptive Damping

Damping (or, equivalently, WSP) plays an important role in HF, as it is needed to ensure that the update stays close enough to $\mathbf{w}_0$ for the quadratic approximation to remain accurate. This gives us a criterion for adapting

λ: it should be large enough to keep the quadratic approximation accurate, but no larger. We can achieve this by monitoring the **reduction ratio**:

$$\rho = \frac{\mathcal{J}(\mathbf{w}) - \mathcal{J}(\mathbf{w}')}{\mathcal{Q}_{\text{quad}}(\mathbf{w}) - \mathcal{Q}_{\text{quad}}(\mathbf{w}')}, \tag{14}$$

where $\mathbf{w}$ and $\mathbf{w}'$ are the old and new iterates, $\mathcal{J}$ is the exact cost function (not including the WSP term), and $\mathcal{Q}_{\text{quad}}$ is the quadratic approximation to the proximal objective (i.e. including the WSP term). The numerator is the true reduction in the cost. The denominator is the reduction in the quadratic objective, plus the WSP term. Consider how ρ behaves in various situations:

- If λ is very large, then $\mathbf{w}'$ is close enough to $\mathbf{w}$ that the *first-order* approximation to $\mathcal{J}$ is accurate, i.e. $\mathcal{J}(\mathbf{w}') - \mathcal{J}(\mathbf{w}) \approx \nabla \mathcal{J}(\mathbf{w})^\top (\mathbf{w}' - \mathbf{w})$. The minimum of the quadratic objective is approximately $\mathbf{w} - \lambda^{-1} \nabla \mathcal{J}(\mathbf{w})$, and a bit of arithmetic shows that $\rho \approx 2$. In this situation, the updates are overly conservative, so we'd certainly like to reduce λ.

- Now suppose we're making a larger update, but the second-order approximation remains accurate. Then the denominator is approximately $\mathcal{J}(\mathbf{w}) - \mathcal{J}(\mathbf{w}') - \frac{\lambda}{2}\|\mathbf{w}' - \mathbf{w}\|^2$. Hence, ρ will generally be between 1 and 2. Since the quadratic approximation remains accurate, we can probably get away with reducing λ.

- If we move far enough that even the second-order approximation is no longer accurate, then the numerator and the denominator may be very different. In the case where $\rho > 1$, the true reduction is larger than we expected, so we can just count ourselves lucky. But if $\rho < 1$, this means we got a smaller reduction than we expected, and if $\rho < 0$, the update actually *increased* the cost. If ρ is much smaller than 1, then we should be more conservative and increase λ.

To summarize, if ρ is close to or larger than 1, then we should decrease λ, whereas if ρ is much smaller than 1, we should increase it. This is captured by the **Levenberg-Marquardt heuristic**:

- If $\rho < \frac{1}{4}$, then $\lambda \leftarrow \frac{3}{2}\lambda$
- If $\rho > \frac{3}{4}$, then $\lambda \leftarrow \frac{2}{3}\lambda$
- Otherwise, λ is unchanged.

Of course, the particular values here can be adjusted, but these values work pretty well as a default.

5 Kronecker-Factored Approximate Curvature

HF is an elegant optimization algorithm, but its use of MVPs to approximate curvature has two fundamental limitations that aren't easily fixed: (1) it approximates the curvature using a single batch, and (2) each weight update requires an expensive iterative procedure (CG). The first problem

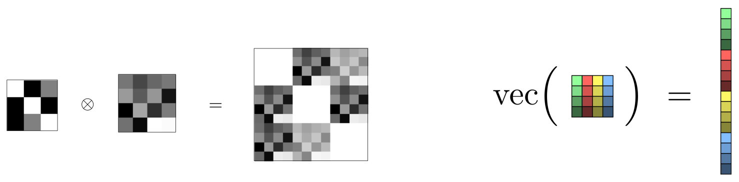

Figure 2: Illustrations of the Kronecker product **(left)** and vectorization operator **(right)**.

means that HF (and other closely related algorithms such as TRPO) often require large batch sizes to be effective, thereby reducing their data throughput. The second problem means HF needs to do a lot more work for a given batch size. This means using HF over SGD only pays off when the convergence benefits of CG are substantial enough to give an orders-of-magnitude reduction in the number of weight updates. This raises the question: can we use second-order approximation, but keep the memory and per-iteration compute costs to within a small multiple of SGD?

The way we'll do this is to fit a parametric approximation to the pullback metric $\mathbf{G}$. In Chapter 3, we already saw that this can be done using the Pullback Sampling Trick (PST). Recall that we had a procedure for sampling vectors $\mathcal{D}\mathbf{w}$ called pseudo-gradients whose covariance is $\mathbf{G}$. In that lecture, we fit a diagonal approximation to $\mathbf{G}$ using the empirical variances of individual pseudo-derivatives $\mathcal{D}w_j$. But a diagonal approximation is very crude, and we can do much better.

Approximating $\mathbf{G}$ with a diagonal matrix is equivalent to fitting the maximum likelihood estimate of a Gaussian distribution with diagonal covariance, i.e. where all of the dimensions are assumed to be independent. But the field of probabilistic graphical models has given us a powerful set of tools for imposing more fine-grained independence assumptions on a set of random variables, so that we can capture important correlations while keeping essential computational operations (such as inversion) tractable. This leads to an algorithm called **Kronecker-Factored Approximate Curvature (K-FAC)** which captures more fine-grained structure of neural net computations.

5.1 Kronecker Product

K-FAC, as suggested by the name, depends crucially on a mathematical operation called the **Kronecker product**, denoted $\mathbf{A} \otimes \mathbf{B}$ for matrices $\mathbf{A}$ and $\mathbf{B}$. This is an operation which concatenates many copies of $\mathbf{B}$, each one scaled by the corresponding entry of $\mathbf{A}$. I.e.,

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & & a_{2n}\mathbf{B} \\ \vdots & & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{pmatrix} \tag{15}$$

The Kronecker product is illustrated in Figure 2.

The reason the Kronecker product is useful is that it allows us to describe operations on matrices in terms of operations on vectors. Recall the Kro-

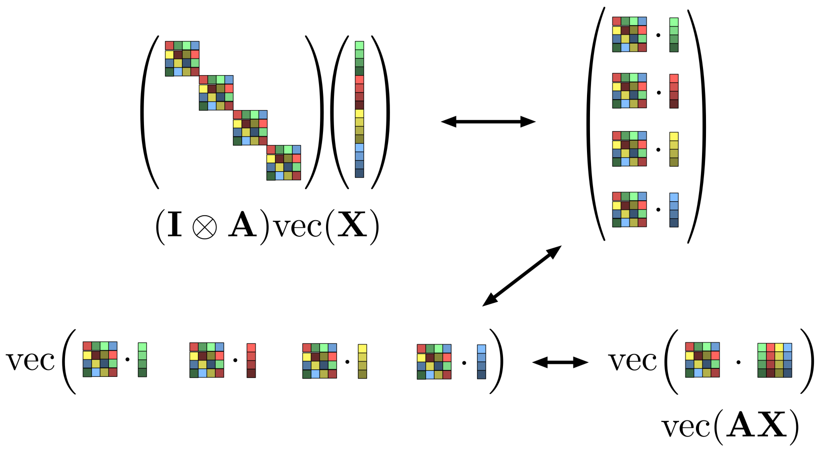
$(\mathbf{I} \otimes \mathbf{A})\mathrm{vec}(\mathbf{X})$
vec
vec
$\mathrm{vec}(\mathbf{AX})$

Figure 3: Proof-by-picture of the identity $\mathrm{vec}(\mathbf{AX}) = (\mathbf{I} \otimes \mathbf{A})\,\mathrm{vec}(\mathbf{X})$, a special case of the more general identity $\mathrm{vec}(\mathbf{AXB}) = (\mathbf{B}^\top \otimes \mathbf{A})\,\mathrm{vec}(\mathbf{X})$ (Eqn. 16).

necker vectorization operator, denoted $\mathrm{vec}(\mathbf{A})$, which stacks the columns of a matrix into a vector (see Figure 2). The Kronecker product allows us to view matrix multiplication as a matrix-vector product using the following important identity:

$$\mathrm{vec}(\mathbf{AXB}) = (\mathbf{B}^\top \otimes \mathbf{A})\,\mathrm{vec}(\mathbf{X}). \tag{16}$$

A proof-by-picture of a special case of this identity is given in Figure 3.

We now list some useful properties of the Kronecker product, all of which are straightforward to derive from Eqn. 16.

1. Matrix multiplication:

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD} \tag{17}$$

2. Matrix transpose:

$$(\mathbf{A} \otimes \mathbf{B})^\top = \mathbf{A}^\top \otimes \mathbf{B}^\top \tag{18}$$

3. Matrix inversion:

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \tag{19}$$

4. Vector outer products:

$$\mathrm{vec}(\mathbf{u}\mathbf{v}^\top) = \mathbf{v} \otimes \mathbf{u}, \tag{20}$$

   where $\mathbf{u}$ and $\mathbf{v}$ are column vectors.

5. If $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are orthogonal, then so is $\mathbf{Q}_1 \otimes \mathbf{Q}_2$.

6. If $\mathbf{D}_1$ and $\mathbf{D}_2$ are diagonal, then so is $\mathbf{D}_1 \otimes \mathbf{D}_2$.

7. If $\mathbf{A}$ and $\mathbf{B}$ are symmetric, then so is $\mathbf{A} \otimes \mathbf{B}$.

8. If $\mathbf{A}$ and $\mathbf{B}$ are symmetric, then the spectral decomposition of $\mathbf{A} \otimes \mathbf{B}$ is given by:

$$\mathbf{A} \otimes \mathbf{B} = (\mathbf{Q_A} \otimes \mathbf{Q_B})(\mathbf{D_A} \otimes \mathbf{D_B})(\mathbf{Q_A^\top} \otimes \mathbf{Q_B^\top}), \tag{21}$$

   where $\mathbf{A} = \mathbf{Q_A}\mathbf{D_A}\mathbf{Q_A^\top}$ and $\mathbf{B} = \mathbf{Q_B}\mathbf{D_B}\mathbf{Q_B^\top}$ are the spectral decompositions of $\mathbf{A}$ and $\mathbf{B}$. Observe that the first and last terms are orthogonal, and the middle one is diagonal, based on the properties listed above. This implies that the eigenvalues of $\mathbf{A} \otimes \mathbf{B}$ consist of all products $\lambda_i \nu_j$ where λ_i and ν_j are eigenvalues of $\mathbf{A}$ and $\mathbf{B}$, respectively. The corresponding eigenvectors are given by $\mathbf{r}_i \otimes \mathbf{s}_j$, where $\mathbf{r}_i$ and $\mathbf{s}_j$ are the eigenvectors of $\mathbf{A}$ and $\mathbf{B}$.

9. If $\mathbf{A}$ and $\mathbf{B}$ are symmetric and positive (semi)definite, then so is $\mathbf{A} \otimes \mathbf{B}$.

5.2 Kronecker-Factored Approximation

To build a probabilistic model to approximate the distribution of the pseudo-gradients $\mathcal{D}\mathbf{w}$, we need to think about the mechanics of backpropagation. Consider a multilayer perceptron for image classification, where each layer $\ell = 1, \ldots, L$ uses the activation function ϕ. It's convenient to use the **homogeneous vector** notation $\bar{\mathbf{a}}_\ell = (\mathbf{a}_\ell^\top \ 1)^\top$ and $\bar{\mathbf{W}}_\ell = (\mathbf{W}_\ell \ \mathbf{b}_\ell)$. For each layer, the forward pass computes a linear transformation followed by the activation function:

$$\begin{aligned} \mathbf{s}_\ell &= \bar{\mathbf{W}}_\ell \bar{\mathbf{a}}_{\ell-1} \\ \mathbf{a}_\ell &= \phi_\ell(\mathbf{s}_\ell) \end{aligned}$$

Once we've computed the logits $\mathbf{z} = \mathbf{a}_L$, we sample $\mathrm{d}\mathbf{z}$ from a distribution whose covariance is $\mathbf{G_z}$. We then compute the activation derivatives and weight derivatives using backprop:

$$\begin{aligned} \mathcal{D}\mathbf{a}_\ell &= \mathbf{W}_\ell^\top \mathcal{D}\mathbf{s}_{\ell+1} \\ \mathcal{D}\mathbf{s}_\ell &= \mathcal{D}\mathbf{a}_\ell \odot \phi'_\ell(\mathbf{s}_\ell) \\ \mathcal{D}\bar{\mathbf{W}}_\ell &= \mathcal{D}\mathbf{s}_\ell \bar{\mathbf{a}}_{\ell-1}^\top \end{aligned}$$

All of this can be summarized in the computation graph shown in Figure 4, where the edges denote which values are directly used to compute which other values. (You can think of this as like the computation graph built by an autodiff framework, except that a lot of nodes are omitted to avoid clutter.)

The exact distribution is intractable to compute with. To make it tractable, we need to simplify the relationships between different variables, either by chopping off edges entirely, or by replacing them with a functional form that's easy to compute with (such as linear). While most of the edges in the original graph are deterministic, we will now treat them as stochastic in order to absorb the errors introduced by simplifying the form of the dependency.

Note that our treatment of modeling error as stochastic noise is no different in principle from any other kind of probabilistic modeling: when we choose to treat an effect as noise, we are not claiming it is inherently stochastic down to the level of physics; rather, we are choosing not to model the mechanism in detail. Even if the universe were purely Newtonian (and hence deterministic), probabilistic models would still be useful descriptions at a higher level.

The most basic — and most commonly used — approximation is to treat all of the layers as independent. (We'll see later how this constraint can be relaxed.) This means the pseudo-derivatives $\mathrm{d}w_i$ and $\mathrm{d}w_j$ are uncorrelated

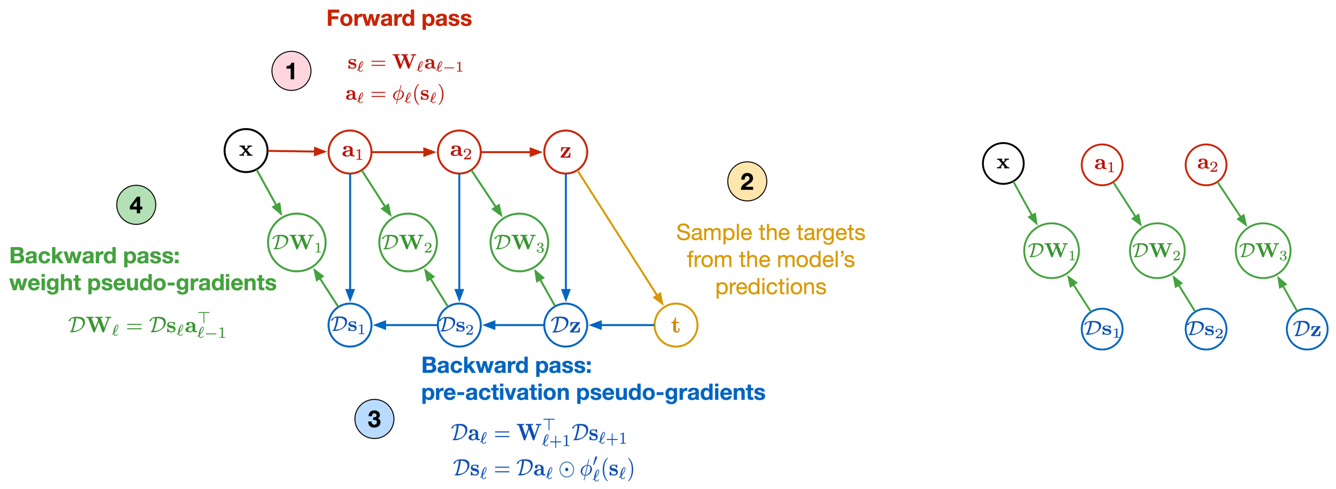
Forward pass
1
$\mathbf{s}_\ell = \mathbf{W}_\ell \mathbf{a}_{\ell-1}$
$\mathbf{a}_\ell = \phi_\ell(\mathbf{s}_\ell)$
2
Sample the targets from the model's predictions
3
Backward pass: pre-activation pseudo-gradients
$\mathcal{D}\mathbf{a}_\ell = \mathbf{W}_{\ell+1}^\top \mathcal{D}\mathbf{s}_{\ell+1}$
$\mathcal{D}\mathbf{s}_\ell = \mathcal{D}\mathbf{a}_\ell \odot \phi'_\ell(\mathbf{s}_\ell)$
4
Backward pass: weight pseudo-gradients
$\mathcal{D}\mathbf{W}_\ell = \mathcal{D}\mathbf{s}_\ell \mathbf{a}_{\ell-1}^\top$
x
$\mathbf{a}_1$
$\mathbf{a}_2$
z
t
$\mathcal{D}\mathbf{W}_1$
$\mathcal{D}\mathbf{W}_2$
$\mathcal{D}\mathbf{W}_3$
$\mathcal{D}\mathbf{s}_1$
$\mathcal{D}\mathbf{s}_2$
$\mathcal{D}\mathbf{z}$

Figure 4: Approximating the distribution of pseudo-gradients for a multilayer perceptron. **(left)** The true computation graph, assuming that the output metric is the Fisher information matrix (see Chapter 3). We are interested in the covariance of the weight pseudo-gradients (green). **(right)** Approximating the activations and pre-activation pseudo-gradients for different layers as independent. This corresponds to removing edges from the probabilistic model, rendering the weight pseudo-gradients for different layers independent. It gives a block-diagonal approximation to the covariance, with one block per layer, and Kronecker-factored blocks.

whenever w_i and w_j belong to different layers. Equivalently, $\mathbf{G}$ is approximated as block diagonal, with one block for each layer of the network. In general, block diagonal approximations are useful because inversion simply requires inverting each of the diagonal blocks. But we're not home free yet, because these blocks are still huge. E.g., a fully connected layer with 1000 input and 1000 output units would require 1 million weights, so each of the diagonal blocks has dimension 1 million.

In order to achieve tractability, we need to impose more structure. Observe that the block of $\mathbf{G}$ corresponding to layer ℓ is the covariance of the weights, flattened into a vector:

$$\begin{aligned}
\mathbf{G}_{\ell\ell} &= \mathbb{E}[\mathrm{vec}(\mathcal{D}\mathbf{W}_\ell)\,\mathrm{vec}(\mathcal{D}\mathbf{W}_\ell)^\top] \\
&= \mathbb{E}[\mathrm{vec}(\mathcal{D}\mathbf{s}_\ell \bar{\mathbf{a}}_{\ell-1}^\top)\,\mathrm{vec}(\mathcal{D}\mathbf{s}_\ell \bar{\mathbf{a}}_{\ell-1}^\top)^\top] \\
&= \mathbb{E}[(\bar{\mathbf{a}}_{\ell-1} \otimes \mathcal{D}\mathbf{s}_\ell)(\bar{\mathbf{a}}_{\ell-1} \otimes \mathcal{D}\mathbf{s}_\ell)^\top] \\
&= \mathbb{E}[\bar{\mathbf{a}}_{\ell-1}\bar{\mathbf{a}}_{\ell-1}^\top \otimes \mathcal{D}\mathbf{s}_\ell \mathcal{D}\mathbf{s}_\ell^\top]
\end{aligned}$$

While this expectation doesn't simplify any further in general, it does simplify if we impose some additional structure on the distribution. Specifically, we approximate the activations $\{\bar{\mathbf{a}}_\ell\}$ as independent of the pre-activation pseudo-gradients $\{\mathcal{D}\mathbf{s}_\ell\}$. Graphically, we can view this as chopping off a lot of edges from the graph, as shown in Figure 4. If $\bar{\mathbf{a}}_{\ell-1}$ is independent of $\mathcal{D}\mathbf{s}_\ell$, then we can push the expectation inward:

The identity that $\mathbb{E}[\mathbf{A} \otimes \mathbf{B}] = \mathbb{E}[\mathbf{A}] \otimes \mathbb{E}[\mathbf{B}]$ when $\mathbf{A}$ and $\mathbf{B}$ are independent random matrices generalizes the well-known fact that $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$ for independent random variables X and Y.

$$\begin{aligned}
\hat{\mathbf{G}}_{\ell\ell} &= \mathbb{E}[\bar{\mathbf{a}}_{\ell-1}\bar{\mathbf{a}}_{\ell-1}^\top] \otimes \mathbb{E}[\mathcal{D}\mathbf{s}_\ell \mathcal{D}\mathbf{s}_\ell^\top] \\
&= \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell,
\end{aligned}$$

where $\mathbf{A}_\ell$ and $\mathbf{S}_\ell$ denote the following covariance matrices:

$$\begin{aligned}\mathbf{A}_\ell &= \mathbb{E}[\bar{\mathbf{a}}_\ell \bar{\mathbf{a}}_\ell^\top] \\ &= \begin{pmatrix} \mathbb{E}[\mathbf{a}_\ell \mathbf{a}_\ell^\top] & \mathbb{E}[\mathbf{a}_\ell] \\ \mathbb{E}[\mathbf{a}_\ell^\top] & 1 \end{pmatrix} \\ \mathbf{S}_\ell &= \mathbb{E}[\mathcal{D}\mathbf{s}_\ell \mathcal{D}\mathbf{s}_\ell^\top]\end{aligned}$$

The fact that we arrive at a Kronecker-factored approximation of each block justifies the name of the algorithm.

How does this help us? First of all, we have a compact way to *represent* $\hat{\mathbf{G}}$. We only need to store the diagonal blocks, and each block is determined by $\mathbf{A}_{\ell-1}$ and $\mathbf{S}_\ell$. If the layer has M input units and N output units, then these matrices are $M \times M$ and $N \times N$, respectively. This is vastly more compact than explicitly storing $\mathbf{G}_{\ell\ell}$, which is an $MN \times MN$ matrix. Note that $\mathbf{W}_\ell$ itself is an $N \times M$ matrix, so as long as $M \approx N$, our Kronecker-factored approximation requires about twice as much storage as the network itself.

Occasionally, we encounter layers where M and N differ by an order of magnitude and one of them is very large. For instance, this occurs in the first fully connected layer of AlexNet. Cases like this require further approximations, but this is beyond the scope of this lecture.

The second way the Kronecker-factored approximation helps us is that solving the linear system is tractable. Suppose we are interested in computing $\hat{\mathbf{G}}^{-1}\mathbf{v}$ for a vector $\mathbf{v}$ of the same size as $\mathbf{w}$. (For instance, $\mathbf{v} = \nabla \mathcal{J}(\mathbf{w})$ when computing the natural gradient.) Let $\bar{\mathbf{V}}_\ell$ denote the entries of $\mathbf{v}$ for layer ℓ, reshaped to match $\bar{\mathbf{W}}_\ell$, and $\mathbf{v}_\ell = \mathrm{vec}(\bar{\mathbf{v}}_\ell)$. Because $\hat{\mathbf{G}}$ is block diagonal, each layer can be computed independently as $\hat{\mathbf{G}}_{\ell\ell}^{-1}\mathbf{v}_\ell$. Applying the properties of the Kronecker product,

$$\begin{aligned}\hat{\mathbf{G}}_{\ell\ell}^{-1}\mathbf{v}_\ell &= (\mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell)^{-1} \mathrm{vec}(\bar{\mathbf{V}}_\ell) \\ &= (\mathbf{A}_{\ell-1}^{-1} \otimes \mathbf{S}_\ell^{-1}) \mathrm{vec}(\bar{\mathbf{V}}_\ell) \\ &= \mathrm{vec}(\mathbf{S}_\ell^{-1} \bar{\mathbf{V}}_\ell \mathbf{A}_{\ell-1}^{-1}).\end{aligned} \tag{22}$$

Computationally, this requires inverting an $M \times M$ matrix and an $N \times N$ matrix, with cost $\mathcal{O}(M^3 + N^3)$. Then it requires matrix multiplications with cost $\mathcal{O}(M^2N + MN^2)$. The cost of the ordinary forward and backward passes are $\mathcal{O}(MNB)$, where B is the batch size. So if we squint, we can say that the K-FAC operations have similar complexity to ordinary backprop. (But the computational overhead is not negligible, and Section 5.5 discusses some strategies for reducing the computational overhead.)

5.3 Damping

In the preceding section, we saw how to compute $\hat{\mathbf{G}}^{-1}\mathbf{v}$, but practical effectiveness of second-order optimizers usually requires damping, so we also need to be able to compute $(\hat{\mathbf{G}} + \lambda \mathbf{I})^{-1}\mathbf{v}$. Fortunately, there's a neat trick that lets us do this. If $\mathbf{C}$ is a symmetric matrix with spectral decomposition $\mathbf{QDQ}^\top$, then $(\mathbf{C} + \lambda\mathbf{I})^{-1} = \mathbf{Q}(\mathbf{D} + \lambda\mathbf{I})^{-1}\mathbf{Q}^\top$. So applying Eqn. 21, we have:

$$(\hat{\mathbf{G}}_{\ell\ell} + \lambda\mathbf{I})^{-1} = (\mathbf{Q}_\mathbf{A} \otimes \mathbf{Q}_\mathbf{S})(\mathbf{D}_\mathbf{A} \otimes \mathbf{D}_\mathbf{S} + \lambda \mathbf{I}_M \otimes \mathbf{I}_N)^{-1}(\mathbf{Q}_\mathbf{A}^\top \otimes \mathbf{Q}_\mathbf{S}^\top),$$

where $\mathbf{A}_{\ell-1} = \mathbf{Q}_\mathbf{A}\mathbf{D}_\mathbf{A}\mathbf{Q}_\mathbf{A}^\top$, $\mathbf{S}_\ell = \mathbf{Q}_\mathbf{S}\mathbf{D}_\mathbf{S}\mathbf{Q}_\mathbf{S}^\top$, and $\mathbf{I}_M$ denotes the $M \times M$ identity matrix. Since $(\hat{\mathbf{G}}_{\ell\ell} + \lambda\mathbf{I})^{-1}$ is a product of three matrices, we compute $(\hat{\mathbf{G}}_{\ell\ell} + \lambda \mathbf{I})^{-1}\mathbf{v}_\ell$ by multiplying by each of these matrices in turn:

$$\begin{aligned}
\bar{\mathbf{V}}'_\ell &= \mathbf{Q}_\mathbf{S}^\top \bar{\mathbf{V}}_\ell \mathbf{Q}_\mathbf{A} \\
[\bar{\mathbf{V}}''_\ell]_{ij} &= \frac{[\bar{\mathbf{V}}'_\ell]_{ij}}{[\mathbf{D}_\mathbf{S}]_{ii}[\mathbf{D}_\mathbf{A}]_{jj} + \lambda} \quad \text{for all } i, j \\
\bar{\mathbf{V}}'''_\ell &= \mathbf{Q}_\mathbf{S} \bar{\mathbf{V}}''_\ell \mathbf{Q}_\mathbf{A}^\top
\end{aligned} \tag{23}$$

Here, $(\hat{\mathbf{G}}_{\ell\ell} + \lambda \mathbf{I})^{-1}\mathbf{v}_\ell = \mathrm{vec}(\bar{\mathbf{V}}'''_\ell)$.

If it's somehow awkward or costly to compute eigendecompositions, there's an alternative damping approach which is only approximate, but only requires inverses. Observe that for any scalar π,

The scalar π can be chosen to minimize the norm of the approximation error, as described by Martens and Grosse (2015)

$$\begin{aligned}
\left(\mathbf{A}_{\ell-1} + \pi\sqrt{\lambda}\mathbf{I}\right) \otimes \left(\mathbf{S}_\ell + \frac{\sqrt{\lambda}}{\pi}\mathbf{I}\right) &= \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell + \pi\sqrt{\lambda}\mathbf{I} \otimes \mathbf{S}_\ell \\
&\quad + \frac{\sqrt{\lambda}}{\pi}\mathbf{A}_{\ell-1} \otimes \mathbf{I} + \lambda \mathbf{I} \otimes \mathbf{I} \\
&\succeq \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell + \lambda \mathbf{I} \otimes \mathbf{I}.
\end{aligned} \tag{24}$$

Recall that $\succeq$ refers to the PSD partial order over symmetric matrices, and the inequality holds because the two terms that are dropped are both PSD. The damped natural gradient can be computed using Eqn. 22, but with the damped versions of $\mathbf{A}_{\ell-1}$ and $\mathbf{S}_\ell$ substituted in. Because of the inequality in Eqn. 24, the update that uses the factored damping is *more conservative* than the update that uses exact damping (Eqn. 23), because it stretches the update by a strictly smaller factor along each of the eigendirections. This is an appealing property, because it means that the worst that can happen is that we take a more cautious step than we would using the exact update.

5.4 Estimating the Covariance Matrices

We need to somehow estimate the Kronecker factors $\{\mathbf{A}_\ell\}$ and $\{\mathbf{S}_\ell\}$, which represent the covariances of the activations and pre-activation pseudo-gradients for each layer. If $\mathbf{Y}$ and $\mathbf{Z}$ denote the matrices of activations and pre-activations for a batch of size B, then the empirical covariances for the batch are given by $\frac{1}{B}\mathbf{Y}_\ell^\top \mathbf{Y}_\ell$ and $\frac{1}{B}\mathcal{D}\mathbf{Z}_\ell^\top \mathcal{D}\mathbf{Z}_\ell$. We'd like to use as much data as possible to estimate the covariances, but we'd also like to avoid using stale estimates if the weights have changed significantly. A good compromise is to maintain exponential moving averages of the statistics:

Sorry for using random letters to denote the activations and pre-activations, but I really didn't have any good options here.

To understand the parameter η, observe that the time constant for the exponential decay is approximately $(1-\eta)^{-1}$, e.g., if $\eta = 0.99$, then we are averaging the statistics over approximately the last 100 batches. A good default value is $\eta = 0.95$.

$$\begin{aligned}
\hat{\mathbf{A}}_\ell &\leftarrow \eta \hat{\mathbf{A}}_\ell + \frac{1-\eta}{B}\mathbf{Y}_\ell^\top \mathbf{Y}_\ell \\
\hat{\mathbf{S}}_\ell &\leftarrow \eta \hat{\mathbf{S}}_\ell + \frac{1-\eta}{B}\mathcal{D}\mathbf{Z}_\ell^\top \mathcal{D}\mathbf{Z}_\ell.
\end{aligned} \tag{25}$$

I should emphasize again that $\hat{\mathbf{S}}_\ell$ is the covariance of the *pseudo-gradients* sampled using the PST, not the covariance of the actual gradients seen during training. (The latter would give the *empirical Fisher matrix*, which is very different from the true Fisher matrix. We'll investigate the empirical Fisher matrix in Chapter 5.)

5.5 Reducing the Computational Overhead

I claimed earlier that K-FAC requires only a small constant factor overhead per iteration relative to ordinary SGD. To evaluate whether this is true, consider the additional work required by K-FAC:

1. **Updating the covariance statistics.** This requires sampling the pseudo-gradients using the PST, as well as some additional matrix multiplications (Eqn. 25).

   Sampling with the PST requires one additional backwards pass (since the forward pass can be reused from the gradient computation). Ordinary backprop requires 2 passes, so according to our crude pass-based accounting scheme, the inclusion of the PST induces 50% overhead. The matrix multiplications in Eqn. 25 require $\mathcal{O}(M^2B)$ and $\mathcal{O}(N^2B)$ operations, compared with $\mathcal{O}(MNB)$ for the forward and backward passes. So assuming $M \approx N$, this also induces a constant factor overhead.

   This source of computational overhead can be mitigated by updating the covariances periodically, or by updating them on a smaller batch than the one used to compute the gradient. Either option induces a tradeoff between computational and statistical efficiency, but in practice the computational overhead of this step can be made fairly small without significantly hindering convergence.

2. **Computing inverses or eigenvalues,** as described in Section 5.3. These operations are both $\mathcal{O}(M^3)$ and $\mathcal{O}(N^3)$, but unfortunately neither operation exploits GPU efficiency as well as matrix multiplications (which otherwise dominate the computational cost). Hence, the wall-clock overhead of this step can be substantial. Between the two options, eigendecompositions can be several times more expensive than inverses.

   As with the covariance updates, we can mitigate the overhead by recomputing the inverses or eigendecompositions only occasionally, for instance once every 20 iterations. Fortunately, the covariances seem to be reasonably stable throughout training (except at the very beginning), so we can get away with these periodic updates without much of a convergence hit. So the overhead from this step can also be made small in practice.

3. **Computing the natural gradient update.** Finally, to compute the approximate natural gradient using Eqn. 22, we need two more matrix multiplications, which have complexity $\mathcal{O}(M^2N)$ and $\mathcal{O}(N^2M)$.

   For fully connected networks, this overhead is substantial if M and/or N is much larger than B (as is usually the case). Unfortunately, we can't solve it with periodic computation like we did in the previous two cases, as the natural gradient needs to be computed in every iteration. Martens and Grosse (2015) observed that if we plug the formula for the weight gradient into Eqn. 22 and simplify, we obtain an alternative formula for the natural gradient whose matrix multiplications instead require $\mathcal{O}(M^2B)$ and $\mathcal{O}(N^2B)$ operations, a substantial improvement.

If we compute the natural gradient using eigendecompositions rather than inverses (Eqn. 23), this requires 4 matrix multiplications rather than 2, so the overhead is doubled.

Unfortunately, this solution breaks the abstraction barrier of the gradient computation, so it can't take advantage of autodiff functionality, and it's generally less flexible than the method described above.

But most of the architectures we're interested in are not fully connected. For the most commonly used architectures (conv nets, RNNs, and transformers), the cost of forward and backward passes is substantially more expensive relative to the number of parameters, while the overhead of the approximate natural gradient computation is still $\mathcal{O}(M^2N)$ and $\mathcal{O}(N^2M)$. Therefore, the overhead of this step is not too bad in practice, and doesn't require any special consideration.

The upshot of all this is that the K-FAC update can be substantially more expensive than the SGD update, but with a bit of attention to computational considerations, the overhead can be easily reduced to a small constant factor (e.g. 1.5x) without substantially hindering convergence. My recommendation is to use a profiler and set the covariance and inverse update intervals large enough to make the overhead acceptably small, and (for most architectures) not to worry about point (3).

One particularly problematic case is when either M or N is unusually large for a given layer. (Only one of them can be very large, otherwise the memory cost for the parameters would be unreasonably large.) For instance, this occurs in the first fully connected layer of AlexNet, or in embedding layers of RNNs and transformers. This needs to be handled on a case-by-case basis by introducing further factorizations Ba et al. (2017), or by reverting to a diagonal approximation to $\mathbf{G}_\ell$ for that layer.

5.6 Putting This All Together

Based on the previous discussion, we can define a basic K-FAC optimizer for fully connected networks. Each iteration requires computing the natural gradient update, and we periodically update the covariances and inverses/eigendecompositions. The most important hyperparameters to tune are the learning rate α and damping parameter λ. The update intervals can be tuned using a profiler (see Section 5.5), and other hyperparameters can be set to reasonable defaults. The full algorithm is summarized in Algorithm 1.

5.7 Extensions

All of the preceding discussion describes the "vanilla" version of K-FAC, i.e. the features which are common to all K-FAC optimizers. It is possible to extend the vanilla algorithm in various ways:

1. **Momentum and iterate averaging.** Heavy ball momentum and iterate averaging are two extremely simple and inexpensive modifications to gradient-based updates which can often substantially speed up convergence. These methods are deferred to Chapter 9 (because their analysis connects naturally to subsequent topics in the course), but are straightforward to add to K-FAC.

2. **Matrix-vector products.** It is possible to make use of exact curvature information for the current mini-batch using MVPs. This can

Algorithm 1: The vanilla K-FAC optimizer.

```
Initialize $\mathbf{w}$ in the usual way;
Estimate the covariance statistics from a large batch of data;
Compute $\mathbf{Q_A}$, $\mathbf{D_A}$, $\mathbf{Q_S}$, $\mathbf{D_S}$ (see Section 5.3);
while not converged do
    Sample a batch of training examples;
    Compute $\nabla\mathcal{J}(\mathbf{w})$ on this batch using backprop;
    if updating covariances this iteration then
        Sample $\{\mathcal{D}\bar{\mathbf{W}}_\ell\}$ using the PST;
        Update the covariance statistics using Eqn. 25;
    if updating eigendecompositions this iteration then
        Compute $\mathbf{Q_A}$, $\mathbf{D_A}$, $\mathbf{Q_S}$, $\mathbf{D_S}$ (see Section 5.3);
    Compute the approximate natural gradient
     $\tilde{\nabla}\mathcal{J}(\mathbf{w}) = \hat{\mathbf{G}}^{-1}\nabla\mathcal{J}(\mathbf{w})$ using Eqn. 23;
    $\mathbf{w} \leftarrow \mathbf{w} - \alpha\tilde{\nabla}\mathcal{J}(\mathbf{w})$;
```

help because MVPs use the *exact* Hessian or pullback metric (albeit only for a single batch), whereas the Kronecker-factored approximation is only an approximation. Among the other benefits, this gives automatic ways to choose the step size and momentum decay parameters in each iteration, reducing the need for hyperparameter tuning. This is briefly explained in Section 6, but see Martens and Grosse (2015) for details.

3. **Beyond block-diagonal.** The preceding discussion assumes layerwise independence, which gives a block-diagonal approximation to $\mathbf{G}$. We can actually do much better. The activations $\mathbf{a}_\ell$ are computed directly from $\mathbf{a}_{\ell-1}$, but depend on activations of layers below that only indirectly through $\mathbf{a}_{\ell-1}$. Similarly, the pseudo-gradients $\mathcal{D}\mathbf{s}_\ell$ are computed directly from $\mathcal{D}\mathbf{s}_{\ell+1}$, but depend on pseudo-gradients for subsequent layers only indirectly through $\mathcal{D}\mathbf{s}_{\ell+1}$. This sort of dependency structure can be modeled probabilistically using a Gaussian graphical model and, somewhat surprisingly, the structure of this graphical model leads to updates which are not much more expensive than those of the block-diagonal approximation given above. This improved approximation seems to give faster convergence than the block-diagonal approximation. However, it is more cumbersome to implement, and it never seems to give spectacular practical gains, so practical K-FAC implementations usually stick to the block-diagonal approximation. See Martens and Grosse (2015) for details.

4. **Other architectures.** The preceding discussion is limited to fully connected layers, but Kronecker-factored approximations have been developed for convolution layers (Grosse and Martens, 2016) and recurrent architectures (Martens et al., 2018). Together, this covers the layer types needed for most of the commonly used architectures.

5. **Distributed implementation.** K-FAC can make use of standard GPU acceleration for neural nets. However, we can parallelize it fur-

ther by distributing the gradient computation across many workers, allowing us to efficiently compute gradients on much larger batches. This sort of parallelism is particularly well-suited to K-FAC, because second-order optimizers can benefit more from large batches (for reasons explained in Chapter 7), and because the primary computational overhead of K-FAC (estimating the covariances and computing inverses or eigenvalues) can be done asynchronously from the gradient updates. See Ba et al. (2017) for details.

5.8 Implementing the Pullback Sampling Trick

Almost the entire K-FAC algorithm can be implemented in more or less orthodox JAX style. The one part that requires a little creativity is the PST sampling, which is used to compute $\{\mathcal{D}\mathbf{s}_\ell\}$. The challenge is that `grad` and `vjp` have a functional API, so they only compute gradients with respect to the inputs to a function. They don't give a convenient way to compute gradients with respect to *intermediate* quantities (such as the pre-activations), since this would break the abstraction barrier.

If we want to recover the activation gradients for particular layers, one feature we'll certainly need is the ability to refer to those layers by name. So we'll define a `named_serial` class, whose API parallels that of `stax.serial`, except that each layer is given a name. For instance, we would like to be able to define an MLP autoencoder architecture as follows:

```
net_init, net_apply = kfac_util.named_serial(
  ('enc1s', Dense(1000)),
  ('enc1a', elementwise(nn.sigmoid)),
  ('enc2s', Dense(500)),
          # etc.
  ('dec3a', elementwise(nn.sigmoid)),
  ('out', Dense(784)),
)
```

Then the pre-activations for the first encoder layer would be referenced as `'enc1s'`, the activations for the first encoder layer as `'enc1a'`, and so on.

Since `vjp` uses a functional API, how do we get at gradients with respect to intermediate values? We need to instrument the VJP computation. The trick is to take in an additional argument consisting of dummy matrices that get *added* to the values that are computed. Since we are not actually interested in modifying the forward pass computations, we will in fact pass in matrices of zeros. However, when we call `vjp`, the gradient with respect to the dummy matrix added to $\mathbf{s}_\ell$ will be exactly the pseudo-gradient $\mathcal{D}\mathbf{s}_\ell$. So we basically need to modify `apply_fn` function in `named_serial` to take this dummy argument. (Some irrelevant features are omitted for clarity.)

```
def named_serial(*layers):
    # ...

    def apply_fn(params, inputs, add_to={}):
        for fun, name in zip(apply_fns, names):
            inputs = fun(params[name], inputs)
            if name in add_to:
                inputs = inputs + add_to[name]

        return inputs

    # ...
```

Now we define the instrumented VJP. The following function returns a VJP operator which takes in the output pseudo-gradients (i.e. $\mathcal{D}\mathbf{z}$) and returns the pseudogradients with respect to the activations of all layers in the network (even though we only require a subset of them). (The `arch` parameter is a `namedtuple` containing functions for things like forward passes.)

```
def make_instrumented_vjp(arch, params, inputs):
    # Compute the activations on a dummy batch to determine the layer sizes.
    dummy_input = np.zeros((2,) + inputs.shape[1:])
    _, dummy_activations = arch.compute_activations(params, dummy_input)

    # Create dummy matrices of zeros the same size as the activations.
    batch_size = inputs.shape[0]
    add_to = {name: np.zeros((batch_size,) + dummy_activations[name].shape[1:])
              for name in dummy_activations}

    # Compute the VJP with respect to the dummy inputs.
    apply_wrap = lambda a: arch.net_apply(params, inputs, a)
    primals_out, vjp_fn = vjp(apply_wrap, add_to)
    return primals_out, vjp_fn
```

The pseudo-gradient covariances for a batch can then be computed as follows. The parameter `output_model` defines the output layer metric (see the diagonal PST implementation in Chapter 3). The final argument `rng` is the key for the random number generator.

```
def estimate_covariances(arch, output_model, w, X, rng):
    logits, vjp_fn = make_instrumented_vjp(arch, arch.unflatten(w), X)
    logit_pseudograds = output_model.sample_pseudograds_fn(logits, rng)
    pseudograds = vjp_fn(logit_pseudograds)[0]

    S = {}
    batch_size = X.shape[0]
    for in_name, out_name in arch.param_info:
        Ds = pseudograds[out_name]
        S[out_name] = Ds.T @ Ds / batch_size

    return S
```

6 Comparing and Combining the Approximations

We've just considered two very different ways to approximate $\mathbf{G}$ and to approximately solve linear systems $\mathbf{G}^{-1}\mathbf{v}$. The first strategy (used in HF) approximates $\mathbf{G}$ using the exact matrix for a single batch of data, and approximately computes $\mathbf{G}^{-1}\mathbf{v}$ using MVPs. This has the advantage that it uses the exact curvature for the batch, but the drawbacks that it only uses a single batch and it requires many CG iterations to compute the update. The second strategy is to fit a parametric approximation to $\mathbf{G}$ (such as K-FAC). The advantages are that it aggregates curvature information over many batches and that inversion is cheap. The disadvantages are that the curvature is only approximate, and that the methods are much less generic to the choice of architecture.

Since the two curvature approximations have complementary strengths and weaknesses, it is natural to combine them. The K-FAC approximation can be thought of as a preconditioner: it provides a cheap operation which implicitly transforms the space to one which is better conditioned. So it would be natural to use K-FAC as a preconditioner for HF.

A more lightweight way to combine the approximations is to use MVPs to choose the step size for K-FAC. After computing the approximate natural gradient $\mathbf{r} = \tilde{\nabla}\mathcal{J}(\mathbf{w})$, we can choose the step size to minimize the quadratic approximation to the cost. Letting $\bar{\mathbf{J}}$ and $\bar{\mathbf{H}}_\mathbf{z}$ denote the Jacobian and output Hessian for the current batch, the second-order Taylor approximation to the cost is given by:

$$\mathcal{J}_{\text{quad}}(\mathbf{w} + \alpha\mathbf{r}) = \mathcal{J}(\mathbf{w}) + \alpha\nabla\mathcal{J}(\mathbf{w})^\top\mathbf{r} + \alpha^2\mathbf{r}^\top\bar{\mathbf{J}}^\top\bar{\mathbf{H}}_\mathbf{z}\bar{\mathbf{J}}\mathbf{r},$$

a quadratic expression in α with minimum given by:

$$\alpha = -\frac{\nabla\mathcal{J}(\mathbf{w})^\top\mathbf{r}}{\mathbf{r}^\top\bar{\mathbf{J}}^\top\bar{\mathbf{H}}_\mathbf{z}\bar{\mathbf{J}}\mathbf{r}}. \tag{26}$$

The most obvious implementation of the step size selection involves an MVP with $\mathbf{G}$, which requires two passes. But we can cut this down to one pass by computing $\mathbf{q} = \bar{\mathbf{J}}\mathbf{r}$ and then computing $\mathbf{q}^\top\bar{\mathbf{H}}_\mathbf{z}\mathbf{q}$.

This quadratic approximation is generally pretty accurate, especially if adaptive damping is used (see Section 4.1). Therefore, this method approximately chooses a step size which minimizes the loss on the current batch. This choice of step size is not necessarily optimal in the stochastic setting (stochastic optimization is discussed in Chapter 7). However, it does address an important problem with fixed step sizes (learning rates), namely that the optimization can get unstable if the learning rate is slightly too large. Because α is chosen to minimize the loss on the current batch, that means it must decrease the loss on that batch, and therefore the optimization cannot blow up.

This instability is, in general, a big problem in tuning learning rates. The reason you might not have encountered it is that most modern architectures include normalization layers, which have the side effect of automatically controlling the step size. See Chapter 5.

The above trick can be generalized to give a sort of momentum effect. In particular, Eqn. 26 can be seen as minimizing the quadratic cost over a 1-dimensional subspace spanned by the approximate natural gradient $\tilde{\nabla}\mathcal{J}(\mathbf{w})$. We can instead minimize the cost over a 2-dimensional subspace spanned by $\tilde{\nabla}\mathcal{J}(\mathbf{w})$ *as well as the previous update vector*:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha\tilde{\nabla}\mathcal{J}(\mathbf{w}) + \beta(\mathbf{w}^{(k)} - \mathbf{w}^{(k-1)}). \tag{27}$$

Can you see how to determine α and β using only two JVPs and no backward passes?

If α and β were fixed values, this equation would simply be heavy ball momentum, and β would be the momentum decay parameter. Hence, the MVPs give a way to automaically adapt the learning rate and momentum decay parameters. Another interesting and surprising interpretation of this update rule is that, in full batch mode close to the optimum, it approximates the CG algorithm. Our discussion of adapting α and β was necessarily brief since we haven't yet discussed momentum, but the full details can be found in Martens and Grosse (2015).

References

J. Ba, R. Grosse, and J. Martens. Distributed second-order optimization using Kronecker-factored approximations. In *International Conference on Learning Representations*, 2017.

Dimitri P. Bertsekas. *Nonlinear Programming.* Athena Scientific, 2016.

R. Grosse and J. Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning*, 2016.

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

J. Martens, J. Ba, and M. Johnson. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

James Martens. Deep learning via Hessian-free optimization. In *International Conference on Machine Learning*, 2010.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer, 2006.

David Pfau, James S. Spencer, Alexander G. D. G. Matthews, and W. M. C. Foulkes. Ab-initio solution of the many-electron Schrödinger equation with deep neural networks. arXiv:1909.02487, 2019.

John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *International Conference on Machine Learning*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv:1707.06347, 2017.

Yuhuai Wu, Elman Mansimov, Roger Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. In *Neural Information Processing Systems*, 2017.