Chapter 3: Metrics

Roger Grosse

1 Introduction

Last week, we looked at the Hessian, which defines the second-order Taylor approximation to the cost function in neural network training. This week, we'll look at a different but closely related type of second-order Taylor approximation. Here, we're interested in approximating some notion of *distance* between two weight vectors. One way to measure distance is using Euclidean distance in weight space (which there's no need to approximate). But we can define lots of other interesting distances, and taking a second-order Taylor approximation to such a distance gives a *metric*.

To motivate why we are interested in metrics, consider the Rosenbrock function (Figure 1), which has been used for decades as a toy problem in optimization:

$$h(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2$$

This function is hard to optimize because its minimum is surrounded by a narrow and flat valley, in which gradient descent gets stuck (Figure 1).

One interesting way of looking at the Rosenbrock function is that it's really the composition of two functions: a nonlinear mapping, and squared Euclidean distance:

$$\begin{aligned}
\mathcal{J}(x_1, x_2) &= \mathcal{L}(f(x_1, x_2)) \\
f(x_1, x_2) &= (a - x_1, \sqrt{b}(x_2 - x_1^2)) \\
\mathcal{L}(z_1, z_2) &= z_1^2 + z_2^2.
\end{aligned}$$

Here, we refer to (x_1, x_2) as the *input variables*, and (z_1, z_2) as the *output variables*. The loss $\mathcal{L}$ is just squared Euclidean distance. It's the nonlinear transformation that makes the optimization difficult. Figure 1(a,b) shows the optimization trajectory in input space and output space, respectively. This general setup, where we are trying to minimze a composition of functions, is known as *composite optimization*.

This setup is roughly analogous to neural net training: (x_1, x_2) are like the weights of the network, and (z_1, z_2) are like the outputs of the network. Typically, we assign a simple, convex loss function in output space, such as squared error or cross-entropy. It's the nonlinear mapping between weights and outputs that makes the optimization difficult. The main place the analogy breaks down is that the neural net cost function is summed over multiple training examples, and the nonlinear mapping (from weights to outptuts) is different each time.

The output loss ($\mathcal{L}$ as a function of (z_1, z_2)) is probably the easiest function in the world to optimize. Figure 1(c,d) shows what would happen if we

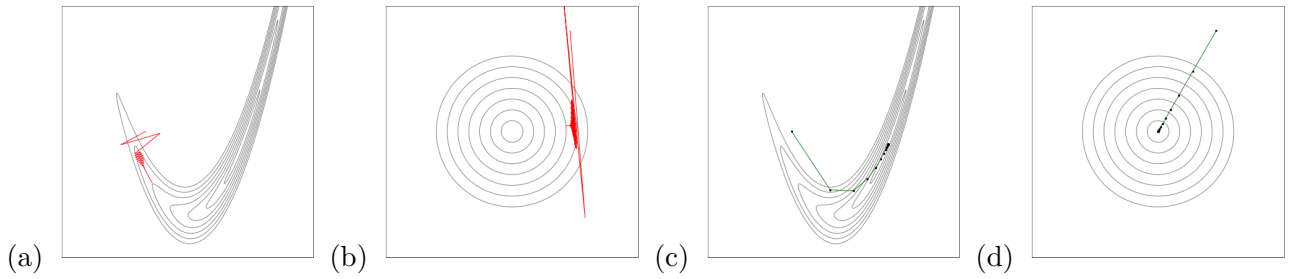


Figure 1: Gradient descent on the Rosenbrock function. **(a)** Gradient descent in input space bounces across the valley and makes slow progress towards the optimum, due to ill-conditioning. **(b)** The corresponding trajectory in output space. **(c,d)** The parameter space and output space trajectories for gradient descent on the outputs.

could do gradient descent directly on the outputs (z_1, z_2). For this particular function, "gradient descent on the outputs" at least makes conceptual sense because the mapping f is invertible. The optimization trajectory darts directly for the optimum (and, in fact, would reach it exactly in 1 step if we used a step size of 1). This is cheating, of course, but it provides us with an idealized optimizer we can try to approximate.

Unfortunately, this notion of "gradient descent on the outputs" doesn't apply to lots of other situations we care about, such as neural net training. One problem is that the neural net function is generally not invertible: there might not be any weight vector that produces a given set of outputs on the training set, or there might be many such weight vectors. In the latter case, it generally won't be easy to *find* such a weight vector. Even if we could, the fact that it performs well on the training set gives us no guarantee about its generalization performance.

However, there's another way to define "gradient descent on the outputs" which *does* carry over to neural nets. In particular, we'll consider *proximal optimization*, where we minimize the cost function (or an approximation therof), plus a *proximity term* which penalizes how far we've moved from the previous iterate. Proximal optimization is especially interesting if we measure proximity in *function space*. Approximating function space proximity gives rise to a useful class of matrices known as *pullback metrics*.

As part of this investigation, we'll look at how to measure dissimilarity between probability distributions. A natural dissimilarity measure between distributions is KL divergence, and taking the second-order Taylor approximation gives us the ubiquitous *Fisher information matrix*. When the distributions are exponential families, this gives rise to an especially beautiful set of mathematical identities.

2 Proximal Optimization

We'll now turn to a class of optimization algorithms known as **proximal methods**. This refers to a general class of methods where in each step we minimize a function plus a *proximity term* which penalizes the distance from the current iterate. While there are some practical optimization algorithms that explicitly use proximal updates, we'll instead use proximal optimiza-

tion as a conceptual tool for thinking about optimization algorithms. We'll start by defining idealized proximal updates which do something interesting but are impractical to compute, and then figure out how to efficiently approximate those updates.

Suppose we're trying to minimize a function $\mathcal{J}(\mathbf{w})$. The idealized update rule, known as the **proximal point method**, is as follows:

$$\mathbf{w}^{(k+1)} = \text{prox}_{\mathcal{J},\lambda}(\mathbf{w}^{(k)}) = \arg\min_{\mathbf{u}} \left[\mathcal{J}(\mathbf{u}) + \lambda\rho(\mathbf{u}, \mathbf{w}^{(k)})\right], \tag{1}$$

where ρ is a **dissimilarity function** which, intuitively, measures the distance between two vectors, but doesn't need to satisfy all the axioms of a distance metric. Canonical examples include squared Euclidean distance and KL divergence. The term $\lambda\rho(\mathbf{u}, \mathbf{w}^{(k)})$ is called the **proximity term**, and the operator $\text{prox}_{f,\lambda}$ is called the **proximal operator**. To make this more concrete, let's consider some specific examples.

To begin with, let's measure dissimilarity using squared Euclidean distance:

$$\rho(\mathbf{u}, \mathbf{v}) = \tfrac{1}{2}\|\mathbf{u} - \mathbf{v}\|^2.$$

Plugging this into Eqn. 1, our proximal operator is given by:

$$\text{prox}_{\mathcal{J},\lambda}(\mathbf{w}^{(k)}) = \arg\min_{\mathbf{u}} \left[\mathcal{J}(\mathbf{u}) + \tfrac{\lambda}{2}\|\mathbf{u} - \mathbf{w}^{(k)}\|^2\right]. \tag{2}$$

If $\mathcal{J}$ is differentiable and convex and $\lambda > 0$, then the proximal objective is strongly convex, so we can find the optimum by setting the gradient to $\mathbf{0}$:

$$\nabla\mathcal{J}(\mathbf{u}_\star) + \lambda(\mathbf{u}_\star - \mathbf{w}^{(k)}) = \mathbf{0}.$$

Rearranging terms, we get an interesting interpretation of the optimality condition:

$$\text{prox}_{\mathcal{J},\lambda}(\mathbf{w}^{(k)}) = \mathbf{u}_\star = \mathbf{w}^{(k)} - \lambda^{-1}\nabla\mathcal{J}(\mathbf{u}_\star). \tag{3}$$

This resembles the gradient descent update for $\mathcal{J}$, except that the gradient is computed at the *new* iterate rather than the *old* one. This equation is not an explicit formula for $\mathbf{u}_\star$ because $\mathbf{u}_\star$ appears on the right-hand side; hence, it is known as the **implicit gradient descent** update.

Clearly it can't be very easy to compute the implicit gradient descent update, since setting $\lambda = 0$ drops the proximity term, so $\text{prox}_{\mathcal{J},0}$ simply minimizes $\mathcal{J}$ directly. Hence, exactly solving the proximal objective in general is as hard as the original optimization problem.

We can make some approximations, however. The first such approximation is to linearize the cost. Returning for a moment to the general proximal objective (Eqn. 1), suppose we linearize $\mathcal{J}$ around the current iterate $\mathbf{w}^{(k)}$:

$$\begin{aligned}\text{prox}_{\mathcal{J},\lambda}(\mathbf{w}^{(k)}) &= \arg\min_{\mathbf{u}} \left[\mathcal{J}(\mathbf{w}^{(k)}) + \nabla\mathcal{J}(\mathbf{w}^{(k)})^\top(\mathbf{u} - \mathbf{w}^{(k)}) + \lambda\rho(\mathbf{u}, \mathbf{w}^{(k)})\right] \\ &= \arg\min_{\mathbf{u}} \left[\nabla\mathcal{J}(\mathbf{w}^{(k)})^\top\mathbf{u} + \lambda\rho(\mathbf{u}, \mathbf{w}^{(k)})\right]\end{aligned}$$

Setting the gradient to 0, we get the following optimality conditions:

$$\nabla_{\mathbf{u}}\rho(\mathbf{u}_\star, \mathbf{w}^{(k)}) = -\lambda^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)}).$$

For some choices of ρ, this gives an algorithm known as **mirror descent** — a term whose meaning will later become clear. For squared Euclidean distance, $\nabla_{\mathbf{u}}\rho(\mathbf{u},\mathbf{v}) = \mathbf{u} - \mathbf{v}$, so mirror descent reduces to the ordinary gradient descent update:

$$\mathbf{u}_\star = \mathbf{w}^{(k)} - \lambda^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)}).$$

A second approach is to take the infinitesimal limit by letting $\lambda \to \infty$. If the proximity term is weighted extremely heavily, then $\mathbf{u}_\star$ will remain close to $\mathbf{w}^{(k)}$. Hence, $\mathcal{J}$ will be well-approximated by the first-order Taylor approximation, just as above. Now consider approximating ρ. First of all, since $\rho(\mathbf{u},\mathbf{v})$ is minimized when $\mathbf{u} = \mathbf{v}$, we have $\nabla_{\mathbf{u}}\rho(\mathbf{u},\mathbf{v})\big|_{\mathbf{u}=\mathbf{v}} = \mathbf{0}$. Hence, we approximate it with the *second-order* Taylor approximation,

$$\rho(\mathbf{u},\mathbf{w}^{(k)}) = \tfrac{1}{2}(\mathbf{u}-\mathbf{w}^{(k)})^\top\mathbf{G}(\mathbf{u}-\mathbf{w}^{(k)}) + \mathcal{O}(\|\mathbf{u}-\mathbf{w}^{(k)}\|^3)$$

where $\mathbf{G} = \nabla^2_{\mathbf{u}}\rho(\mathbf{u},\mathbf{w}^{(k)})\big|_{\mathbf{u}=\mathbf{w}^{(k)}}$ is known as the **metric matrix**. This formula can also be written in terms of a class of distance metrics called **Mahalanobis distance**, which you can visualize as a stretched-out version of Euclidean distance:

$$\rho(\mathbf{u},\mathbf{w}^{(k)}) = \tfrac{1}{2}\|\mathbf{u}-\mathbf{w}^{(k)}\|^2_{\mathbf{G}} + \mathcal{O}(\|\mathbf{u}-\mathbf{w}^{(k)}\|^3)$$
$$\|\mathbf{v}\|_{\mathbf{G}} = \sqrt{\mathbf{v}^\top\mathbf{G}\mathbf{v}}.$$

Plugging both approximations into Eqn. 1, we get:

$$\mathrm{prox}_{\mathcal{J},\lambda}(\mathbf{w}^{(k)}) = \arg\min_{\mathbf{u}}\left[\nabla\mathcal{J}(\mathbf{w}^{(k)})^\top\mathbf{u} + \tfrac{\lambda}{2}(\mathbf{u}-\mathbf{w}^{(k)})^\top\mathbf{G}(\mathbf{u}-\mathbf{w}^{(k)})\right],$$

with optimum

$$\mathbf{u}_\star = \mathbf{w}^{(k)} - \lambda^{-1}\mathbf{G}^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)}). \tag{4}$$

This optimal solution resembles the Newton update $\mathbf{w}^{(k)} - \alpha\mathbf{H}^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)})$, except that $\mathbf{H}$ (the Hessian of $\mathcal{J}$) is replaced by $\mathbf{G}$ (the Hessian of ρ). In the case where ρ represents squared Euclidean distance, $\mathbf{G} = \mathbf{I}$, so this solution *also* reduces to the ordinary gradient descent update. In general, however, we'll see that these two approximations can yield interestingly different algorithms.

Since we're considering the $\lambda \to \infty$ limit, this implies that implicit gradient descent behaves like ordinary gradient descent when the steps are small enough.

Finally, a third approach is to take a second-order Taylor approximation to $\mathcal{J}$ and a second-order Taylor approximation to ρ. This might be more accurate than the previous approximation for larger step sizes, because we're using a second-order rather than first-order approximation to $\mathcal{J}$. The update rule, derived similarly to the ones above, is given by:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - (\mathbf{H} + \lambda\mathbf{G})^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)}). \tag{5}$$

If ρ is chosen to be squared Euclidean distance, then $\mathbf{G} = \mathbf{I}$, and this gives a **damped Newton** update. Intuitively, the damping term prevents the optimizer from taking a very large step in low curvature directions, perhaps helping to stabilize the optimization. We won't consider this particular approximation any further in today's lecture, but we'll discuss the role of damping in more detail next week, when we talk about second-order optimization of neural nets.

2.1 Trust Region Methods

Proximal methods are also closely related to another family of optimization algorithms known as **trust region** methods. Here, the soft proximity penalty is converted to a hard constraint:

$$\arg\min_{\mathbf{u}} \mathcal{J}(\mathbf{u}) \qquad \text{s.t.} \qquad \rho(\mathbf{u}, \mathbf{w}^{(k)}) \leq \eta, \tag{6}$$

where η is a hyperparameter controlling how far each update is allowed to wander from the current iterate. If $\mathcal{J}$ is convex, then the trust region problem is actually equivalent to the proximal problem, in the sense that any optimum to Eqn. 6 is also an optimum to Eqn. 1 for some value of λ, and vice versa. The difference between these approaches is simply whether you'd prefer to control the size of the updates directly, or control the weight of the proximity term.

3 Fisher Information

So far, our only example of proximal updates used the Euclidean metric, which isn't that interesting because the results agree with the ordinary gradient. Proximal updates become much more powerful if we can use a more intrinsically meaningful dissimilarity function. In the case of probability distributions, a natural dissimilarity function is **KL divergence**:

$$\mathrm{D}_{\mathrm{KL}}(q \,\|\, p) = \mathbb{E}_{\mathbf{x}\sim q}[\log q(\mathbf{x}) - \log p(\mathbf{x})] \tag{7}$$

KL divergence isn't truly a distance metric, because it it asymmetric and doesn't satisfy the triangle inequality. However, it's very convenient as a way of measuring how different two distributions are. For instance, it has the information theoretic interpretation as *relative entropy*, i.e. the number of bits wasted if you try to encode data from source q using a code designed for another source p. It can be shown that D_{KL} is nonnegative, and $\mathrm{D}_{\mathrm{KL}}(q \,\|\, q) = 0$ for any distribution q — basic properties we need from any dissimilarity function.

Actually, KL divergence is more closely analogous to *squared* Euclidean distance than to Euclidean distance. However, $\sqrt{\mathrm{D}_{\mathrm{KL}}}$ doesn't satisfy symmetry or the triangle inequality either.

For a great tutorial introduction to KL divergence, see `https://colah.github.io/posts/2015-09-Visual-Information/`.

Notice that Eqn. 7 doesn't mention the parameters of the distributions. That's because KL divergence is an **intrinsic** dissimilarity function between distributions, i.e. it doesn't care how they're parameterized. But if we're trying to *learn* a distribution, we'll typically restrict ourselves to some parametric family $\{p_{\boldsymbol{\theta}}\}$ (such as Gaussians), parameterized by $\boldsymbol{\theta}$.

Recall that when we derived approximations to the proximal operators, we sometimes needed the Hessian of the dissimilarity function. For $\rho = \mathrm{D}_{\mathrm{KL}}$, this is given by the **Fisher information matrix**, denoted $\mathbf{F}_{\boldsymbol{\theta}}$. (The subscript indicates the parameterization, but we'll drop it when it's obvious from context.)

Exercise: derive these expressions for $\mathbf{F}_{\boldsymbol{\theta}}$.

$$\begin{aligned} \nabla^2_{\mathbf{u}} \mathrm{D}_{\mathrm{KL}}(p_{\mathbf{u}} \,\|\, p_{\boldsymbol{\theta}})\big|_{\mathbf{u}=\boldsymbol{\theta}} &= \mathbf{F}_{\boldsymbol{\theta}} \\ &= \mathrm{Cov}_{\mathbf{x}\sim p_{\boldsymbol{\theta}}}(\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x})) \\ &= \mathbb{E}_{\mathbf{x}\sim p_{\boldsymbol{\theta}}}\left[(\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x}))(\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x}))^\top\right] \end{aligned} \tag{8}$$

The second and third lines give explicit formulas for $\mathbf{F}_{\boldsymbol{\theta}}$ in terms of the log-likelihood gradients $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$, which are called the **Fisher scores**. The

third equality follows from the general identity (which applies to any random vector $\mathbf{v}$):

$$\mathrm{Cov}(\mathbf{v}) = \mathbb{E}[\mathbf{v}\mathbf{v}^\top] - \mathbb{E}[\mathbf{v}]\mathbb{E}[\mathbf{v}]^\top,$$

combined with the observation that

$$\mathbb{E}_{\mathbf{x}\sim p_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x})] = \mathbf{0}.$$

The fact that $\mathbb{E}_{\mathbf{x}\sim p_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x})] = \mathbf{0}$ can be proved using integration by parts. But the intuition is that the true parameters $\boldsymbol{\theta}$ maximize the expected log-likelihood for data drawn from $p_{\boldsymbol{\theta}}$, so the log-likelihood gradient should be $\mathbf{0}$ in expectation.

The explicit formulas for $\mathbf{F}_{\boldsymbol{\theta}}$ may be more familiar from a statistics class than the interpretation as the Hessian of D_{KL}. But I'd argue that the Hessian of D_{KL} is really the right way to think about $\mathbf{F}_{\boldsymbol{\theta}}$, i.e. it defines a local distance metric in the space of distributions.

So the exact proximal update is as follows:

$$\mathrm{prox}_{\mathcal{J},\lambda}(\boldsymbol{\theta}) = \arg\min_{\mathbf{u}} \left[\mathcal{J}(\mathbf{u}) + \lambda \mathrm{D}_{\mathrm{KL}}(p_{\mathbf{u}} \,\|\, p_{\boldsymbol{\theta}})\right]. \tag{9}$$

Let's take the infinitesimal limit, i.e. $\lambda \to \infty$, given by Eqn. 4. This gives the following update:

$$\begin{aligned} \boldsymbol{\theta}^{(k+1)} &= \boldsymbol{\theta} - \alpha \mathbf{F}_{\boldsymbol{\theta}}^{-1} \nabla \mathcal{J}(\boldsymbol{\theta}) \\ &= \boldsymbol{\theta} - \alpha \tilde{\nabla} \mathcal{J}(\boldsymbol{\theta}), \end{aligned} \tag{10}$$

where $\alpha = \lambda^{-1}$ and the vector $\tilde{\nabla}\mathcal{J}(\boldsymbol{\theta}) = \mathbf{F}_{\boldsymbol{\theta}}^{-1}\nabla\mathcal{J}(\boldsymbol{\theta})$ is the **natural gradient**. This update rule is known as **natural gradient descent**. The term *natural* comes from the fact that the update is independent of how the distribution is parameterized, up to the first order. More on this in Section 6.

4 Exponential Families

The above discussion applies to any parametric family of distributions. But we get some very interesting interpretations of natural gradient when we specialize to particular classes of distributions known as **exponential families**. In general, an exponential family is a family of probability distributions that has the following form:

The discussion of exponential families can be skipped without too much loss of continuity. But it's so beautiful that I just *had* to include it.

$$p_{\boldsymbol{\eta}}(\mathbf{x}) = \frac{h(\mathbf{x})}{\mathcal{Z}(\boldsymbol{\eta})} \exp\left(\boldsymbol{\eta}^\top \mathbf{t}(\mathbf{x})\right), \tag{11}$$

where $\boldsymbol{\eta}$ are the **natural parameters** of the distribution, and $\mathbf{t}$ is a vector of **sufficient statistics**. (We'll see where this term comes from when we discuss maximum likelihood.) The factor $1/\mathcal{Z}(\boldsymbol{\eta})$ is known as the **normalizing constant** because its role is to ensure that the PDF integrates to 1, and $\mathcal{Z}(\boldsymbol{\eta})$ is the **partition function**, defined as

$$\begin{aligned} \mathcal{Z}(\boldsymbol{\eta}) &= \int h(\mathbf{x}) \exp\left(\boldsymbol{\eta}^\top \mathbf{t}(\mathbf{x})\right) \mathrm{d}\mathbf{x} \quad \text{or} \\ \mathcal{Z}(\boldsymbol{\eta}) &= \sum_{\mathbf{x}} h(\mathbf{x}) \exp\left(\boldsymbol{\eta}^\top \mathbf{t}(\mathbf{x})\right), \end{aligned} \tag{12}$$

depending if the distribution is continuous or discrete.

The **moments** of an exponential family distribution are the expected sufficient statistics:

$$\boldsymbol{\xi} = \mathbb{E}_{\mathbf{x}\sim p_{\boldsymbol{\eta}}}[\mathbf{t}(\mathbf{x})].$$

Under some regularity conditions, it can be shown that there is a one-to-one mapping between the moments and the natural parameters. Hence, $\boldsymbol{\xi}$ can be seen as an alternative parameterization of the exponential family, in which case we write $p_{\boldsymbol{\xi}}$.

Now let's see some concrete examples of exponential families.

1. The Bernoulli distribution, where $x \in \{0, 1\}$, and $\Pr(x = 1) = \theta$. This can be seen as an exponential family with $\xi(x) = x$ and $h(x) = 1$. The moment is $\mathbb{E}[\xi(x)] = \theta$. It can be shown that the natural parameters satisfy:

   $$\eta = \log \frac{\theta}{1-\theta} \qquad \theta = \sigma(\eta) = \frac{1}{1+\exp(-\eta)}.$$

   Hence, the natural parameter represents the **logit**, which gives the log odds ratio between the two outcomes. The moment is obtained from the natural parameters using the **logistic function** σ, a standard neural network activation function. It's common to use the logistic activation function for the output layer in binary classification (either with neural nets or logistic regression), so the output pre-activations can be seen as the natural parameters, and the output activations as the moments.

2. The categorical distribution, where $\mathbf{x}$ is a **one-hot vector**, i.e. a binary vector with exactly one entry being 1, and the vector $\boldsymbol{\theta}$ represents the probabilities of each outcome. This is a generalization of the Bernoulli distribution to more than 2 possible outcomes. It can be seen as an exponential family with $\mathbf{t}(\mathbf{x}) = \mathbf{x}$ and $h(\mathbf{x}) = 1$. The moments are $\mathbb{E}[\mathbf{t}(\mathbf{x})] = \boldsymbol{\theta}$. The natural parameters can be defined as:

   $$\eta_i = \log \theta_i \qquad \theta_i = [\boldsymbol{\sigma}(\boldsymbol{\eta})]_i = \frac{\exp(\eta_i)}{\sum_j \exp(\eta_j)}.$$

   The function $\boldsymbol{\sigma}$ is the **softmax function**, the standard activation function for output layers of a classification network. This shows that the natural parameters $\boldsymbol{\eta}$, called the **logits**, can be seen as the log odds ratios of the outcomes.

   This representation is good enough for many purposes. However, it's not a *minimal* exponential family, since the sufficient statistics are linearly dependent (the entries always sum to 1). This can be fixed by truncating the final coordinate of the one-hot vector, so that the Kth class is arbitrarily assigned to $\mathbf{0}$. The formulas are a bit more cumbersome, but basically similar to the ones given above.

3. Now consider a Gaussian distribution with fixed covariance $\boldsymbol{\Sigma}$, parameterized by the mean vector $\boldsymbol{\mu}$:

   $$\begin{aligned} p_\mu(\mathbf{x}) &= \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \\ &= \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\tfrac{1}{2}\mathbf{x}^\top \boldsymbol{\Sigma}^{-1}\mathbf{x} + \boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1}\mathbf{x} - \tfrac{1}{2}\boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}\right) \end{aligned}$$

This is an exponential family distribution with sufficient statistics $\mathbf{t}(\mathbf{x}) = \mathbf{x}$, natural parameters $\mathbf{h} = \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}$ (also called the **potential vector**), and moments $\boldsymbol{\xi} = \mathbb{E}[\mathbf{t}(\mathbf{x})] = \boldsymbol{\mu}$. Hence, in this instance, the natural parameters and sufficient statistics are equivalent, up to a linear transformation. You can check that:

This parameterization might seem a little weird, which just shows that the natural parameters aren't always the most "natural" way to parameterize a distribution.

$$h(\mathbf{x}) = \exp(-\tfrac{1}{2}\mathbf{x}^\top \boldsymbol{\Sigma}^{-1}\mathbf{x})$$
$$\mathcal{Z}(\mathbf{h}) = (2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}\exp(-\tfrac{1}{2}\mathbf{h}^\top \boldsymbol{\Sigma}\mathbf{h})$$

4. Now make the Gaussian zero-mean, but parameterize it by the covariance matrix $\boldsymbol{\Sigma}$. This is an exponential family with sufficient statistics $t(\mathbf{x}) = -\frac{1}{2}\operatorname{vec}(\mathbf{x}\mathbf{x}^\top)$, natural parameters $\boldsymbol{\eta} = \operatorname{vec}(\boldsymbol{\Sigma}^{-1})$, and moments $\boldsymbol{\xi} = \mathbb{E}[\mathbf{t}(\mathbf{x})] = -\frac{1}{2}\operatorname{vec}(\boldsymbol{\Sigma})$. The notation vec denotes the **Kronecker vectorization** operator which stacks the columns of a matrix into a vector. The matrix $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ is called the **precision matrix**; it is a positive definite matrix which, intuitively, is large in the directions where the distribution is the most constrained (i.e. most "precise").

Our formulation of Gaussians here is not a *minimal* exponential family. To make it minimal, we should instead extract the upper triangular entries of $\boldsymbol{\Lambda}$ and $\mathbf{x}\mathbf{x}^\top$.

5. Finally, let's put this together by considering a general Gaussian distribution parameterized by both $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. To make the exponential family interpretation more obvious, we can rewrite the Gaussian PDF in another form called **information form**:

$$p_{\mathbf{h},\boldsymbol{\Lambda}}(\mathbf{x}) = \frac{|\boldsymbol{\Lambda}|^{1/2}}{(2\pi)^{D/2}}\exp\left(-\tfrac{1}{2}\mathbf{x}^\top\boldsymbol{\Lambda}\mathbf{x} + \mathbf{h}^\top\mathbf{x} - \tfrac{1}{2}\mathbf{h}^\top\boldsymbol{\Lambda}^{-1}\mathbf{h}\right).$$

The sufficient statistics, natural parameters, and moments are given by:

$$\mathbf{t}(\mathbf{x}) = \begin{pmatrix}\mathbf{x}\\ -\frac{1}{2}\operatorname{vec}(\mathbf{x}\mathbf{x}^\top)\end{pmatrix} \qquad \boldsymbol{\eta} = \begin{pmatrix}\mathbf{h}\\ \operatorname{vec}(\boldsymbol{\Lambda})\end{pmatrix} \qquad \boldsymbol{\xi} = \begin{pmatrix}\boldsymbol{\mu}\\ -\frac{1}{2}\operatorname{vec}(\boldsymbol{\Sigma} + \boldsymbol{\mu}\boldsymbol{\mu}^\top)\end{pmatrix}$$

Hence, the natural parameters are exactly the information form parameters, reshaped into a vector, and the moments are (proportional to) the first and second moments of the distribution. The moments aren't quite the same as the standard parameterization in terms of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ (which is known as **covariance form**), but each is easily obtainable from the other.

Covariance form and information form are two fundamental parameterizations of the Gaussian distribution. Converting between them requires a matrix inversion (to compute $\boldsymbol{\Lambda}$ from $\boldsymbol{\Sigma}$ or vice versa), which is an $\mathcal{O}(D^3)$ operation. For many applications, this inversion is feasible but expensive, hence we'd like to do it as rarely as possible. Hence, algorithms based on manipulating Gaussian distributions are carefully designed to carry out some steps in information form and some steps in covariance form in order to minimize the number of inversions. This is crucial in, e.g., Kalman filtering and Gaussian belief propagation.

The innocuous-looking partition function $\mathcal{Z}(\boldsymbol{\eta})$ is the source of many elegant identities involving exponential families.

1. First of all, we get a nice formula for the moments:

$$\boldsymbol{\xi} = \nabla \log \mathcal{Z}(\boldsymbol{\eta}). \tag{13}$$

This is nice because we have convenient automatic differentiation tools, but not convenient tools for computing expectations. This identity means we can obtain the moments simply by writing a function that computes $\log \mathcal{Z}(\boldsymbol{\eta})$, and then calling `grad` on it.

2. The formula for the log-likelihood is:

$$\begin{aligned} \ell(\boldsymbol{\eta}) &= \sum_{i=1}^{N} \log p_{\boldsymbol{\eta}}(\mathbf{x}) \\ &= \sum_{i=1}^{N} \left[\boldsymbol{\eta}^\top \mathbf{t}(\mathbf{x}^{(i)}) - \log \mathcal{Z}(\boldsymbol{\eta}) \right], \end{aligned} \tag{14}$$

and its gradient is

$$\begin{aligned} \nabla_{\boldsymbol{\eta}} \log p_{\boldsymbol{\eta}}(\mathbf{x}) &= \mathbf{t}(\mathbf{x}) - \boldsymbol{\xi} \qquad \text{(from Eqn. 13)} \\ \nabla \ell(\boldsymbol{\eta}) &= \hat{\boldsymbol{\xi}} - \boldsymbol{\xi} \\ \hat{\boldsymbol{\xi}} &= \frac{1}{N} \sum_{i=1}^{N} \mathbf{t}(\mathbf{x}^{(i)}), \end{aligned} \tag{15}$$

where $\hat{\boldsymbol{\xi}}$ is called the **empirical moments**. Setting the gradient to $\mathbf{0}$, we find that the likelihood is maximized when $\boldsymbol{\xi} = \hat{\boldsymbol{\xi}}$. This implies that, to compute the maximum likelihood parameters, we only need to store $\hat{\boldsymbol{\xi}}$, and can otherwise forget the data; this is why $\mathbf{t}$ is called the *sufficient statistic*. Since maximum likelihood involves matching the model moments to the empirical moments, this is known as **moment matching**.

You can check that this formula is consistent with the well-known maximum likelihood estimates for Bernoulli, categorical, and Gaussian distributions.

3. We just saw that $\nabla_{\boldsymbol{\eta}} \log p_{\boldsymbol{\eta}}(\mathbf{x}) = \mathbf{t}(\mathbf{x}) - \boldsymbol{\xi}$ (see Eqn. 15). Plugging this into Eqn. 8, we get a convenient formula for the Fisher information matrix:

$$\mathbf{F}_{\boldsymbol{\eta}} = \mathrm{Cov}_{\mathbf{x} \sim p_{\boldsymbol{\eta}}}(\mathbf{t}(\mathbf{x})) \tag{16}$$

4. The KL divergence is given by:

$$\begin{aligned} \mathrm{D}_{\mathrm{KL}}(p_{\boldsymbol{\eta}_1} \,\|\, p_{\boldsymbol{\eta}_2}) &= \mathbb{E}_{\mathbf{x} \sim p_{\boldsymbol{\eta}_1}} [\log p_{\boldsymbol{\eta}_1}(\mathbf{x}) - \log p_{\boldsymbol{\eta}_2}(\mathbf{x})] \\ &= \mathbb{E}_{\mathbf{x} \sim p_{\boldsymbol{\eta}_1}} [\boldsymbol{\eta}_1^\top \mathbf{t}(\mathbf{x}) - \boldsymbol{\eta}_2^\top \mathbf{t}(\mathbf{x})] - \log \mathcal{Z}(\boldsymbol{\eta}_1) + \log \mathcal{Z}(\boldsymbol{\eta}_2). \end{aligned}$$

Taking the Hessian with respect to $\boldsymbol{\eta}_2$, all of the linear and constant terms drop out, and we're left with just $\nabla^2 \log \mathcal{Z}(\boldsymbol{\eta}_2)$. But the Hessian of KL divergence is $\mathbf{F}_{\boldsymbol{\eta}}$ (Eqn. 8). So this gives us a neat identity

$$\mathbf{F}_{\boldsymbol{\eta}} = \nabla^2 \log \mathcal{Z}(\boldsymbol{\eta}). \tag{17}$$

Just like Eqn. 13 gave us a convenient way to compute $\boldsymbol{\xi}$ in an autodiff framework, Eqn. 17 gives us a convenient way to compute $\mathbf{F}$: just take the function that computes $\log \mathcal{Z}(\boldsymbol{\eta})$ and call `grad` on it twice. (Or, if you don't want to construct the full matrix, you can compute MVPs with $\mathbf{F}$ by computing Hessian-vector products with $\log \mathcal{Z}$.)

5. As a consequence of Eqn. 17, the Hessian of the negative log-likelihood is also $\mathbf{F}_{\boldsymbol{\eta}}$:

$$\begin{aligned}\nabla^2_{\boldsymbol{\eta}} \log p_{\boldsymbol{\eta}}(\mathbf{x}) &= \underbrace{\nabla^2_{\eta}[\boldsymbol{\eta}^\top \mathbf{t}(\mathbf{x}^{(i)})]}_{=\mathbf{0}} - \nabla^2_{\boldsymbol{\eta}} \log \mathcal{Z}(\boldsymbol{\eta}) \\ &= -\mathbf{F}_{\boldsymbol{\eta}}.\end{aligned} \tag{18}$$

   This implies that the natural gradient descent update for maximum likelihood estimation in exponential families is the same as the Newton-Raphson update (up to a scale factor).

6. We saw that $\boldsymbol{\xi} = \nabla \log \mathcal{Z}(\boldsymbol{\eta})$ and $\mathbf{F}_{\boldsymbol{\eta}} = \nabla^2 \log \mathcal{Z}(\boldsymbol{\eta})$. But since the Hessian is just the gradient of the gradient, this implies:

$$\mathbf{F}_{\boldsymbol{\eta}} = \nabla \boldsymbol{\xi}(\boldsymbol{\eta}) = \mathbf{J}_{\boldsymbol{\xi},\boldsymbol{\eta}}, \tag{19}$$

   where $\mathbf{J}_{\boldsymbol{\xi},\boldsymbol{\eta}}$ denotes the Jacobian of the mapping from natural parameters to moments. Another way to write this is:

$$\mathrm{d}\boldsymbol{\xi} = \mathbf{F}_{\boldsymbol{\eta}} \mathrm{d}\boldsymbol{\eta},$$

   where $\mathrm{d}\boldsymbol{\xi}$ and $\mathrm{d}\boldsymbol{\eta}$ denote infinitesimal perturbations to the moments and natural parameters, respectively. Note that the Jacobian of the inverse mapping is just $\mathbf{J}_{\boldsymbol{\eta},\boldsymbol{\xi}} = \mathbf{F}_{\boldsymbol{\eta}}^{-1}$. This is a surprisingly useful identity, and shows that the Fisher information matrix fundamentally relates the two coordinate systems.

We've just seen a number of beautiful identities relating two coordinate systems for exponential families — natural parameters and moments — and they all somehow pass through $\log \mathcal{Z}$. This is no accident: the relationship between natural parameters and moments is a special case of **Legendre duality**, and there's a beautiful field called **information geometry** which explores these sorts of ideas. Amari and Nagaoka (2000) is the classic text on this topic, and Amari (2016) is more up-to-date though less polished.

4.1 Proximal Operators in Exponential Families

When we compute the proximal operator for an exponential family using KL divergence as the dissimilarity function, something neat happens. First, suppose we take the first-order approximation to the cost function, but keep the proximal term exact. Then we're minimizing:

In this section, we'll use $\mathcal{J}_{\boldsymbol{\eta}}$ and $\mathcal{J}_{\boldsymbol{\xi}}$ to denote the cost function, viewed as a function of $\boldsymbol{\eta}$ or $\boldsymbol{\xi}$.

$$\nabla \mathcal{J}_{\boldsymbol{\eta}}(\boldsymbol{\eta}^{(k)})^\top \mathbf{u} + \lambda \mathrm{D}_{\mathrm{KL}}(p_{\boldsymbol{\eta}^{(k)}} \,\|\, p_{\mathbf{u}}).$$

Computing the gradient with respect to $\mathbf{u}$ and setting it to $\mathbf{0}$, we find that:

$$\boldsymbol{\xi}^{(k+1)} = \boldsymbol{\xi}^{(k)} - \lambda^{-1} \nabla \mathcal{J}_{\boldsymbol{\eta}}(\boldsymbol{\eta}^{(k)}). \tag{20}$$

In other words, the *moments* are updated opposite the gradient computed for the *natural parameters*!

Similarly, suppose we use the opposite direction of KL divergence for the proximity term. Here, it's more convenient to use the moments parameterization, so let $\tilde{\mathcal{J}}$ denote the cost function parameterized in terms of the moments. Then we're minimizing:

$$\nabla \mathcal{J}_{\boldsymbol{\xi}}(\boldsymbol{\xi}^{(k)})^\top \mathbf{u} + \lambda \mathrm{D}_{\mathrm{KL}}(p_{\mathbf{u}} \,\|\, p_{\boldsymbol{\xi}^{(k)}}).$$

Computing the gradient with respect to $\mathbf{u}$ and setting it to $\mathbf{0}$, we find that:

$$\boldsymbol{\eta}^{(k+1)} = \boldsymbol{\eta}^{(k)} - \lambda^{-1} \nabla \mathcal{J}_{\boldsymbol{\xi}}(\boldsymbol{\xi}^{(k)}). \tag{21}$$

So we update the *natural parameters* opposite the gradient computed for the *moments*! These two update rules, where you compute the gradient in one coordinate system and then apply it in the other coordinate system, are known as **mirror descent**.

Now we show that the natural gradient update is equivalent to mirror descent. By the Chain Rule for derivatives, we have:

You can also see that natural gradient descent is equivalent to mirror descent by taking the limit of Eqns. 20 and 21 as $\lambda \to \infty$.

$$\begin{aligned}
\nabla_{\boldsymbol{\eta}} \mathcal{J}_{\boldsymbol{\eta}}(\boldsymbol{\eta}) &= \nabla_{\boldsymbol{\eta}} \mathcal{J}_{\boldsymbol{\xi}}(\boldsymbol{\xi}(\boldsymbol{\eta})) \\
&= \mathbf{J}_{\boldsymbol{\xi},\boldsymbol{\eta}}^\top \nabla_{\boldsymbol{\xi}} \mathcal{J}_{\boldsymbol{\xi}}(\boldsymbol{\xi}(\boldsymbol{\eta})) \\
&= \mathbf{F}_{\boldsymbol{\eta}} \nabla_{\boldsymbol{\xi}} \mathcal{J}_{\boldsymbol{\xi}}(\boldsymbol{\xi}(\boldsymbol{\eta})) \qquad \text{(Eqn. 19)}
\end{aligned}$$

Multiplying both sides by $\mathbf{F}_{\boldsymbol{\eta}}^{-1}$, we get a surprising formula for the natural gradient:

$$\tilde{\nabla}_{\boldsymbol{\eta}} \mathcal{J}_{\boldsymbol{\eta}}(\boldsymbol{\eta}) = \mathbf{F}_{\boldsymbol{\eta}}^{-1} \nabla_{\boldsymbol{\eta}} \mathcal{J}_{\boldsymbol{\eta}}(\boldsymbol{\eta}) = \nabla_{\boldsymbol{\xi}} \mathcal{J}_{\boldsymbol{\xi}}(\boldsymbol{\xi}(\boldsymbol{\eta})). \tag{22}$$

Similarly, it can be shown that

$$\tilde{\nabla}_{\boldsymbol{\xi}} \mathcal{J}_{\boldsymbol{\xi}}(\boldsymbol{\xi}) = \nabla_{\boldsymbol{\eta}} \mathcal{J}_{\boldsymbol{\eta}}(\boldsymbol{\eta}(\boldsymbol{\xi})). \tag{23}$$

So the natural gradient with respect to the natural parameters is the ordinary Euclidean gradient with respect to the moments, and vice versa!

Why is this connection useful? For one thing, it's often the easiest way to derive the natural gradient update in practice. E.g., suppose you want to compute the natural gradient update for a multivariate Gaussian distribution with unknown mean and covariance. If you're feeling masochistic, you could do this by representing the Gaussian as a *minimal* exponential family (i.e. taking the upper triangular entries of $\boldsymbol{\Sigma}$, etc.) and then somehow deriving an expression for $\mathbf{F}$. Sound fun? Or you can solve it more simply by applying Eqns. 22 or 23.

4.2 Maximum Likelihood and Fisher Efficiency

Now we'll consider a case where natural gradient descent — and, equivalently, mirror descent — clearly does the right thing: maximum likelihood in exponential families. Here, we're trying to maximize the log-likelihood (Eqn. 14), for which the optimal solution is given by moment matching, $\boldsymbol{\xi} = \hat{\boldsymbol{\xi}}$ (Eqn. 15). This can clearly be done by sweeping once over the training set to compute $\hat{\boldsymbol{\xi}}$. But what if we do natural gradient descent instead?

We'll consider the **online learning** setting, where we iterate *once* through the training set, updating on one training example at a time. The log-likelihood gradient for a single example is given by (see Eqn. 15):

$$\nabla_{\boldsymbol{\eta}} \log p_{\boldsymbol{\eta}}(\mathbf{x}) = \mathbf{t}(\mathbf{x}) - \boldsymbol{\xi}$$

By plugging this into Eqns. 20 and 23, we see that the mirror descent update, and the natural gradient update for $\boldsymbol{\xi}$, are both given by:

$$\boldsymbol{\xi}^{(k+1)} = \boldsymbol{\xi}^{(k)} - \alpha_k (\boldsymbol{\xi}^{(k)} - \mathbf{t}(\mathbf{x}^{(k+1)})). \tag{24}$$

I write α_k rather than α so that we can choose a **learning rate schedule**, i.e. a different learning rate for each time step. An interesting choice of schedule is $\alpha_k = 1/k$. You can show by induction that this update computes a running average of the empirical moments:

$$\boldsymbol{\xi}^{(k)} = \frac{1}{k}\sum_{i=1}^{k} \mathbf{t}(\mathbf{x}^{(i)}). \tag{25}$$

In other words, the kth iterate is exactly the batch maximum likelihood estimate of the parameters from the first k examples! As a special case, once the entire dataset is processed, we have the exact maximum likelihood estimate $\boldsymbol{\xi}^{(N)} = \hat{\boldsymbol{\xi}}$.

This is interesting because the batch maximum likelihood estimate is a trick which only applies to exponential families, while mirror descent and natural gradient descent are generic algorithms that can be applied to lots more cost functions. The property that the online algorithms come close to the information theoretic optimum is known as **Fisher efficiency**, and Amari (1998) showed that natural gradient descent is Fisher efficient in a wider variety of situations than just maximum likelihood. In Chapter 7, we'll derive a result closely related to Amari's analysis of natural gradient descent.

The mirror descent and natural gradient updates for maximum likelihood have the property that each update results in a statistically optimal inference from the information seen so far. Algorithms which incrementally update a probabilistic model as new information arrives are known as **filtering** algorithms. Natural gradient descent can be interpreted as a filtering algorithm in a wider variety of situations, although typically the correspondence is only approximate (Ollivier, 2018).

5 Approximating Function Space Proximity

We've just seen two examples of dissimilarity functions: squared Euclidean distance, and KL divergence. Between these, only squared Euclidean distance is directly applicable to neural nets. (KL divergence is for probability distributions, and a neural net isn't a probability distribution.) But Euclidean distance in weight space is unappealing, because it depends on an arbitrary parameterization of the network (see Section 6) and because it neglects the fact that some directions in weight space can have a much larger effect than others on the network's predictions. When dealing with probability distributions, a convenient property of KL divergence is that it is *natural*: it depends only on the distributions themselves, not how they're parameterized. Can we define natural dissimilarity functions for neural nets?

The mathematical operation we'll use to do this is called **pullback**. The general idea is simple: if we have a differentiable map $\mathbf{z} = f(\mathbf{x})$ and a function $g(\mathbf{z}_1, \ldots, \mathbf{z}_K)$ defined on the output space, then the pullback of g over f, denoted f^*g, is defined as follows:

$$f^*g(\mathbf{x}_1, \ldots, \mathbf{x}_K) = g(f(\mathbf{x}_1), \ldots, f(\mathbf{x}_K)). \tag{26}$$

To clarify the notation: f^* is an operator. When we write $f^*g(\cdots)$, this means we first evaluate f^*g (which gives us a function), and then we evaluate this function on the arguments in parentheses.

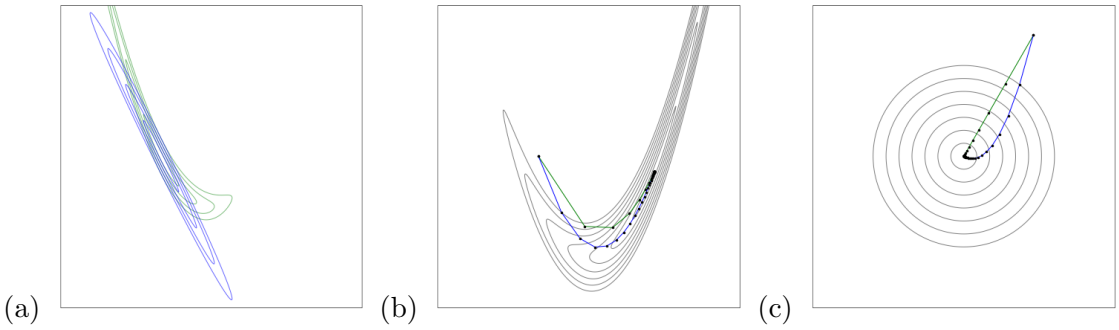


Figure 2: Approximate proximal optimization applied to the Rosenbrock function. **(a)** The exact pullback metric (green) and its second-order approximation (blue), shown in terms of distance from the initialization. **(b,c)** The natural gradient (or Gauss-Newton) trajectory (blue), compared against the exact proximal point trajectory (green), which is equivalent to gradient descent on the outputs.

Before we turn to neural nets, consider the Rosenbrock function example from Section 1. We saw that gradient descent on the inputs has trouble with ill-conditioned curvature caused by the nonlinear mapping. Gradient descent on output space converges immediately (Figure 1), but this solution doesn't generalize to neural nets because it relies on the mapping having a tractable inverse. The trick is: instead of doing *output space gradient descent*, we can instead do gradient descent on the inputs using an *output space dissimilarity function*. This works almost as well, and we'll see that it has a direct analogue for neural nets. Specifically, let $\rho_{\text{euc}}(\mathbf{z}, \mathbf{z}') = \frac{1}{2}\|\mathbf{z}-\mathbf{z}'\|^2$, and consider its pullback to input space:

$$f^*\rho_{\text{euc}}(\mathbf{x}, \mathbf{x}') = \tfrac{1}{2}\|f(\mathbf{x}) - f(\mathbf{x}')\|^2.$$

The resulting dissimilarity function in shown in Figure 2(a). Observe that the proximal point algorithm with this dissimilarity function is equivalent to implicit gradient descent on the outputs. Therefore, by approximating the proximal point update, hopefully we can achieve similar behavior to output space gradient descent.

The proximal point algorithm and implicit gradient descent are explained in Section 2.

We can approximate the pullback of an output space dissimilarity function by taking a second-order Taylor approximation around the current inputs $\mathbf{x}_0$. To derive the second-order Taylor approximation, we need the constant term, the gradient, and the Hessian. The constant term is just

$$f^*\rho(\mathbf{x}_0, \mathbf{x}_0) = \rho(f(\mathbf{x}_0), f(\mathbf{x}_0)) = 0.$$

Using the Chain Rule, the gradient is:

$$\begin{aligned}\nabla_{\mathbf{x}} f^*\rho(\mathbf{x}, \mathbf{x}_0)\big|_{\mathbf{x}=\mathbf{x}_0} &= \mathbf{J}_{\mathbf{z}\mathbf{x}}^\top \underbrace{\nabla_{\mathbf{z}}\rho(\mathbf{z}, \mathbf{z}_0)\big|_{\mathbf{z}=\mathbf{z}_0}}_{=\mathbf{0}} \\ &= \mathbf{0}\end{aligned}$$

Finally, we derive the Hessian using the same decomposition we used in Chapter 2 to get the Gauss-Newton approximation. The difference is that

this time the formula is exact because the second term is $\mathbf{0}$:

$$\begin{aligned}\nabla_{\mathbf{x}}^2 f^*\rho(\mathbf{x}, \mathbf{x}_0)\big|_{\mathbf{x}=\mathbf{x}_0} &= \mathbf{J}_{\mathbf{zx}}^\top \left[\nabla_{\mathbf{z}}^2 \rho(\mathbf{z}, \mathbf{z}_0)\big|_{\mathbf{z}=\mathbf{z}'}\right] \mathbf{J}_{\mathbf{zx}} + \underbrace{\sum_a \frac{\partial \rho}{\partial z_a} \nabla_{\mathbf{x}}^2 [f(\mathbf{x})]_a}_{=\mathbf{0}} \\ &= \mathbf{J}_{\mathbf{zx}}^\top \mathbf{G}_{\mathbf{z}} \mathbf{J}_{\mathbf{zx}},\end{aligned} \tag{27}$$

where $\mathbf{G}_{\mathbf{z}} = \nabla_{\mathbf{z}}^2 \rho(\mathbf{z}, \mathbf{z}_0)|_{\mathbf{z}=\mathbf{z}_0}$ is the output space metric matrix. In the first line, the reason the second term drops out is that it contains the output space partial derivatives $\partial\rho/\partial y_a$, which are 0. Hence, our second-order Taylor approximation to the pullback $f^*\rho(\mathbf{x}, \mathbf{x}_0)$ is the Mahalanobis metric:

$$\begin{aligned}f^*\rho(\mathbf{x}, \mathbf{x}_0) &\approx \tfrac{1}{2}\|\mathbf{x} - \mathbf{x}_0\|_{\mathbf{G}_{\mathbf{x}}}^2 \\ &= \tfrac{1}{2}(\mathbf{x} - \mathbf{x}_0)^\top \mathbf{G}_{\mathbf{x}} (\mathbf{x} - \mathbf{x}_0),\end{aligned}$$

where $\mathbf{G}_{\mathbf{x}} = \mathbf{J}_{\mathbf{zx}}^\top \mathbf{G}_{\mathbf{z}} \mathbf{J}_{\mathbf{zx}}$. In general, we'll drop the subscripts on $\mathbf{G}_{\mathbf{x}}$ when it's clear from context which metric matrix we're referring to.

For our Rosenbrock example, we chose squared Euclidean distance on output space. Hence, $\mathbf{G}_{\mathbf{z}} = \mathbf{I}$, and $\mathbf{G}_{\mathbf{x}} = \mathbf{J}_{\mathbf{zx}}^\top \mathbf{J}_{\mathbf{zx}}$ turns out to equal the Gauss-Newton matrix (see Chapter 2). The natural gradient updates, therefore, are equivalent to the Gauss-Newton updates up to a scale factor:

The connection with the Gauss-Newton Hessian is discussed further in Section 5.1.

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \mathbf{G}_{\mathbf{x}}^{-1} \nabla \mathcal{J}(\mathbf{x}^{(k)}). \tag{28}$$

The natural gradient (or Gauss-Newton) trajectory is shown in Figure 2(b,c). While it doesn't exactly match the output space gradient descent trajectory, it has a qualitatively similar behavior, and in particular doesn't get stuck bouncing across the valley the way gradient descent does (Figure 1(a,b)).

Now let's generalize this to neural nets (and function approximation more generally). Here, we'll measure the dissimilarity between two weight vectors by how different the networks' outputs are *in expectation*. Often it's possible to choose a meaningful dissimilarity function ρ on output space, such as squared Euclidean distance:

$$\rho(\mathbf{z}, \mathbf{z}') = \tfrac{1}{2}\|\mathbf{z} - \mathbf{z}'\|^2.$$

The distance between weight vectors, then, can be measured in terms of the average dissimilarity between the outputs, taken with respect to the data distrubution. This is referred to informally as **function space distance**. Letting $f_{\mathbf{x}}(\mathbf{w}) = f(\mathbf{w}, \mathbf{x})$, we define:

In weight space, ρ_{pull} is only a *semimetric*, not a metric, because it may be 0 for distinct weight vectors. This distinction won't concern us.

$$\rho_{\text{pull}}(\mathbf{w}, \mathbf{w}') = \mathbb{E}_{\mathbf{x}}[f_{\mathbf{x}}^* \rho(\mathbf{w}, \mathbf{w}')] = \mathbb{E}_{\mathbf{x}}[\rho(f(\mathbf{w}, \mathbf{x}), f(\mathbf{w}', \mathbf{x}))].$$

or its finite sample version:

$$\rho_{\text{pull}}(\mathbf{w}, \mathbf{w}') = \frac{1}{N} \sum_{i=1}^N [\rho(f(\mathbf{w}, \mathbf{x}^{(i)}), f(\mathbf{w}', \mathbf{x}^{(i)}))].$$

This notion of function space distance is illustrated in Figure 3.

Now consider the second-order Taylor approximation to ρ_{pull}:

$$\begin{aligned}\nabla_{\mathbf{w}}^2 \rho_{\text{pull}}(\mathbf{w}, \mathbf{w}')\big|_{\mathbf{w}=\mathbf{w}'} &= \mathbb{E}_{\mathbf{x}}\left[\nabla_{\mathbf{w}}^2 \rho(f(\mathbf{w}, \mathbf{x}), f(\mathbf{w}', \mathbf{x}))\right] \\ &= \mathbb{E}_{\mathbf{x}}[\mathbf{J}_{\mathbf{zw}}^\top \mathbf{G}_{\mathbf{z}} \mathbf{J}_{\mathbf{zw}}].\end{aligned} \tag{29}$$

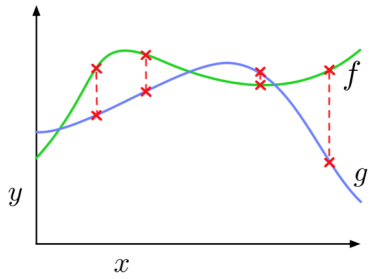


Figure 3: Illustration of function space distance. If we have a 1-D regression problem with 4 training examples, the function space distance between two functions is approximated as the mean of the squared distances between the predictions on these 4 training inputs.

(The derivation follows Eqn. 27). As before, $\mathbf{G}_\mathbf{z} = \nabla^2_\mathbf{z}\rho(\mathbf{z}, \mathbf{z}')$ is the metric matrix for output space.

We denote the weight space metric matrix as $\mathbf{G} = \nabla^2_\mathbf{w}\rho_{\text{pull}}(\mathbf{w}, \mathbf{w}')\big|_{\mathbf{w}=\mathbf{w}'}$ (or $\mathbf{G}_\mathbf{w}$ when we want to distinguish it from other metric matrices) because the letter g is traditionally used to denote metrics in Riemannian geometry. I'll refer to it as the **pullback metric**, because it's constructed using the pullback operation. (This isn't a standard term, but no standard term has yet been adopted.) The choice of the letter $\mathbf{G}$ collides with our notation for the Gauss-Newton Hessian (see Chapter 2), but that's OK because we'll now see that the two matrices are equivalent in a lot of important situations.

For those who are familiar with Riemannian geometry, notice an alternative way to construct $\mathbf{G}_\mathbf{w}$: we could have instead defined a Riemannian metric on output space and pulled it back to weight space over f. This gives another way to understand the term *pullback metric*.

5.1 Connection to the Gauss-Newton Hessian

In our Rosenbrock example, we saw that the pullback metric for Euclidean distance agrees with the classical Gauss-Newton matrix (see Eqn. 28). This is a special case of a more general relationship. Given a strictly convex function ϕ, the **Bregman divergence** between two points $\mathbf{z}$ and $\mathbf{z}'$ is defined as:

$$\mathrm{D}_\phi(\mathbf{z}, \mathbf{z}') = \phi(\mathbf{z}) - \phi(\mathbf{z}') - \nabla\phi(\mathbf{z}')^\top(\mathbf{z} - \mathbf{z}').$$

We can understand this geometrically as follows (see Figure 4). Because ϕ is convex, the first-order Taylor approximation to ϕ at $\mathbf{z}'$ is a lower bound on ϕ. The further a point is from $\mathbf{z}'$, the less accurate the lower bound will be. The Bregman divergence $\mathrm{D}_\phi(\mathbf{z}, \mathbf{z}')$ is simply the gap between $\phi(\mathbf{z})$ and the lower bound.

Like KL divergence, Bregman divergences don't satisfy the requirements to be a distance metric, in particular symmetry and the triangle inequality. However, like KL divergence, they have some convenient properties: for instance they are nonnegative, convex, zero only when $\mathbf{z} = \mathbf{z}'$. Notable examples include squared Euclidean distance (generated by $\phi(\mathbf{z}) = \frac{1}{2}\|\mathbf{z}\|^2$) and KL divergence in an exponential family (generated by $\phi(\boldsymbol{\eta}) = \log \mathcal{Z}(\boldsymbol{\eta})$).

Observe that the Hessian of the Bregman divergence is simply the Hessian of ϕ:

$$\nabla^2_\mathbf{z}\mathrm{D}_\phi(\mathbf{z}, \mathbf{z}')|_{\mathbf{z}=\mathbf{z}'} = \nabla^2\phi(\mathbf{z}'). \tag{30}$$

Recall that the Gauss-Newton Hessian (see Chapter 2) was defined as $\mathbf{G} =$

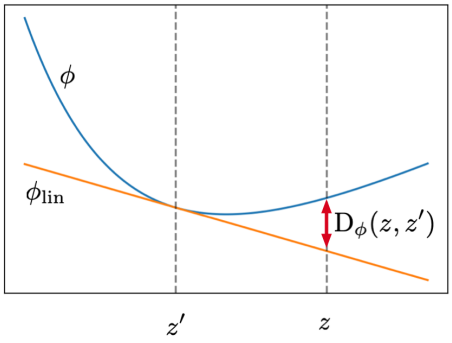


Figure 4: Illustration of Bregman divergence. The Bregman divergence $\mathrm{D}_\phi(\cdot, y')$ is the gap between the convex function ϕ and its first-order Taylor approximation around y'.

$\mathbb{E}_{\mathbf{x}}[\mathbf{J}_{\mathbf{zw}}^\top \mathbf{H}_{\mathbf{z}} \mathbf{J}_{\mathbf{zw}}]$, while the pullback metric is defined as $\mathbf{G}_{\mathbf{w}} = \mathbb{E}_{\mathbf{x}}[\mathbf{J}_{\mathbf{zw}}^\top \mathbf{G}_{\mathbf{z}} \mathbf{J}_{\mathbf{zw}}]$. If the loss function is convex (e.g. squared error, cross-entropy), we can use its generated Bregman divergence as the output dissimilarity function. In this case, we get $\mathbf{G}_{\mathbf{z}} = \mathbf{H}_{\mathbf{z}}$, and the metric matrix $\mathbf{G}_{\mathbf{w}}$ equals the Gauss-Newton Hessian. This equivalence justifies our choice of the letter $\mathbf{G}$ to denote both matrices.

5.2 Computing with Pullback Metrics: The Pullback Sampling Trick

One way to compute with pullback metrics is to use implicit matrix-vector products, just like we did throughout Chapter 2. For some problems, this is indeed the best way to do it. This can be done using a minor variant of our Gauss-Newton HVP from Chapter 2:

```
def pullback_mvp(f, rho, w, v):
    z0, R_z = jvp(f, (w,), (v,))
    rho_z0 = lambda z: rho(z, z0)
    R_gz = hvp(rho_z0, z0, R_z)
    _, f_vjp = vjp(f, w)
    return f_vjp(R_gz)[0]
```

But the MVP approach has the disadvantage that it approximates the metric with a batch of data. Hence, if one wants to compute the matrix-vector product on the full dataset, one needs to do forward passes over the whole dataset. This can be expensive. We now consider an alternative approach, whereby we fit a parametric approximation to the pullback metric, such as a diagonal matrix. This way, we have a compact form for the metric, and don't need to refer to the individual training examples. Furthermore, certain parametric approximations support efficient *inverses*, an operation which can be very expensive when approximated using MVPs and conjugate gradient.

We can fit such a parametric approximation using the **Pullback Sampling Trick (PST)**. (This is not a standard term, but no standard term

has been adopted.) Consider that if a random vector $\mathbf{x}$ has expectation $\mathbf{0}$ and covariance $\mathbf{\Sigma}$, then for a (fixed) matrix $\mathbf{A}$ of appropriate dimensions, the random vector $\mathbf{Ax}$ has expectation $\mathbf{0}$ and covariance $\mathbf{A\Sigma A}^\top$. Therefore, the following procedure samples a zero-mean vector $\mathcal{D}\mathbf{w}$ whose covariance is $\mathbf{G}$:

1. Sample an input $\mathbf{x}$ and compute the outputs $\mathbf{z}$ by doing a forward pass through the network.
2. Sample a vector $\mathcal{D}\mathbf{z}$ (in output space) whose covariance is $\mathbf{G_z}$.
3. Pull it back to weight space by computing $\mathcal{D}\mathbf{w} = \mathbf{J}_{\mathbf{zw}}^\top \mathcal{D}\mathbf{z}$ (i.e. backprop).

So now we have zero-mean random vectors $\mathcal{D}\mathbf{w}$ whose covariance is $\mathbf{G}$. We'll refer to $\mathcal{D}\mathbf{w}$ as the **pseudogradient** because the procedure for computing it closely resembles that of gradient computation. (Again, this is not a standard term, but no standard term has been adopted.)

We'd now like to compactly approximate $\mathbf{G} = \text{Cov}(\mathcal{D}\mathbf{w})$. The most straightforward approximation is to take the diagonal entries, i.e. the empirical variances of the individual entries of $\mathcal{D}\mathbf{w}$. If we fit our approximation $\hat{\mathbf{G}}$ from a finite set of samples, then it will be a diagonal matrix whose diagonal entries are:

$$\hat{\mathbf{G}}_{ii} = \frac{1}{S}\sum_{s=1}^{S} \mathcal{D}w_i^2. \tag{31}$$

This corresponds to the maximum likelihood estimate of an axis-aligned Gaussian distribution.

How can we implement this in JAX? First of all, the API needs some way for the user to specify the output metric $\mathbf{G_z}$. The user does this by providing a function that samples output space vectors whose mean is $\mathbf{0}$ and whose covariance is $\mathbf{G_z}$. For example, for Euclidean distance in output space, we have $\mathbf{G_z} = \mathbf{I}$, so we can simply sample i.i.d. standard normal random variables:

```
def euclidean_metric_sample(z, rng):
    return random.normal(rng, shape=z.shape)
```

In order for our code to exploit our GPU capacity, we'd like to run the above procedure for a batch of examples. Steps 1 and 2 above can be done in batch mode in the usual way. However, in Step 3, we need to separately compute the VJP for each training example, so that we can sum up the squares of each entry. JAX's `vmap` function is very useful in this context: it lets us write code that computes something for a single training example, and it automatically figures out how to compute it in a vectorized way for an entire batch. So what we need to do is write a function that computes the VJP (Step 3) for a single training example, and then call `vmap` on it. Then we can sum the squared gradients in the obvious way.

```
def diag_pst_estimate(net_fn, w, x, output_sample_fn, rng):
    # Sample the output pseudogradient
    z = net_fn(w, x)
    Dz = output_sample_fn(z, rng)
```

```
    # Function that pulls back the output pseudogradient for one example
    def pullback(xi, Dzi):
        # Append a dummy dimension for batch size of 1
        xi_, Dzi_ = xi[np.newaxis, ...], Dzi[np.newaxis, ...]

        # Compute the weight pseudogradient for this "batch"
        net_fn_wrap = lambda w: net_fn(w, xi_)
        _, net_vjp = vjp(net_fn_wrap, w)
        return net_vjp(Dzi_)[0]

    Dw = pullback(x[0,:], Dz[0,:])

    # Compute the matrix of pseudogradients for individual examples, and sum the squares
    pullback_vec = vmap(pullback, in_axes=(0,0), out_axes=0)
    Dw_samples = pullback_vec(x, Dz)
    return np.mean(Dw_samples**2, axis=0)
```

Unfortunately, there's a big problem with this implementation: it's extremely memory-inefficient, because it explicitly constructs the matrix of all the individual pseudogradients. For large modern networks, it's impractical to store more than a handful of gradient vectors in memory. We could call the above code on a very small batch, but then we only get a small number of samples relative to the work that we do. I'm not aware of any generic trick for computing the diagonal PST estimate for arbitrary architectures which is both vectorized and memory efficient. It is possible to implement efficient PST samplers for particular layer types, and the excellent PyTorch package `backpack` does exactly this. (An equivalent package for JAX has yet to be written.) In Chapter 4, we'll see an efficient implementation of another version of the PST which uses a much more accurate approximation than the diagonal one.

The PST estimate can be averaged over multiple batches in order to obtain a more accurate approximation to $\mathbf{G}$. Alternatively, we might want to maintain an online estimate $\hat{\mathbf{G}}$ during training, in which case we might instead update our estimate using an **exponential moving average** with parameter η:

$$\hat{\mathbf{G}}_{ii}^{(k+1)} \leftarrow (1-\eta)\hat{\mathbf{G}}_{ii}^{(k)} + \eta[\mathcal{D}w_i^{(k)}]^2. \tag{32}$$

One way to think about the parameter η is that the timescale of the moving average is $1/\eta$. Hence, $\eta = 0.01$ corresponds to averaging over roughly the past 100 iterations.

5.3 The Fisher Information Matrix for Neural Networks

Many of our neural networks output the natural parameters of an exponential family distribution over the targets, which we'll denote as $r(\cdot \,|\, \mathbf{x})$. For instance, classification is typically done using cross-entropy loss under a Bernoulli or categorical distribution. Least squares regression problems can be viewed as maximizing likelihood under a Gaussian observation model. If the outputs represent a probability distribution, it might make more sense to use KL divergence, rather than squared Euclidean distance, as the output dissimilarity function. As we saw in Section 3, Taylor approximating KL divergence gives rise to the Fisher-Rao metric, represented by the Fisher information matrix.

We can derive the pullback metric for Fisher information, just as we would for any other output space metric. However, we can simplify the formula a little bit. In this derivation, we'll use the shorthand that $\mathcal{D}\mathbf{v} = \nabla_{\mathbf{v}} \log r(\mathbf{t} \,|\, \mathbf{x})$ denotes the log-likelihood gradient with respect to a variable $\mathbf{v}$.

Here, I write $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\cdot]$ to emphasize that $\mathbf{x}$ is sampled from the data distribution. While this was also the case for all the other metrics we've considered, I write it explicitly here in order to contrast it with the sampling procedure for $\mathbf{t}$, which is sampled from the network's predictions.

$$
\begin{aligned}
\mathbf{F}_{\mathbf{w}} &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\mathbf{J}_{\mathbf{zw}}^\top \mathbf{F}_{\mathbf{z}} \mathbf{J}_{\mathbf{zw}}\right] && \text{(Eqn. 29)} \\
&= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\mathbf{J}_{\mathbf{zw}}^\top \mathbb{E}_{\mathbf{t}\sim r(\cdot\,|\,\mathbf{x})}[\mathcal{D}\mathbf{z}\mathcal{D}\mathbf{z}^\top] \mathbf{J}_{\mathbf{zw}}\right] && \text{(Eqn. 8)} \\
&= \mathbb{E}_{\substack{\mathbf{x}\sim p_{\text{data}} \\ \mathbf{t}\sim r(\cdot\,|\,\mathbf{x})}}\left[\mathbf{J}_{\mathbf{zw}}^\top \mathcal{D}\mathbf{z}\mathcal{D}\mathbf{z}^\top \mathbf{J}_{\mathbf{zw}}\right] \\
&= \mathbb{E}_{\substack{\mathbf{x}\sim p_{\text{data}} \\ \mathbf{t}\sim r(\cdot\,|\,\mathbf{x})}}[\mathcal{D}\mathbf{w}\mathcal{D}\mathbf{w}^\top] && \text{(Chain Rule)}
\end{aligned}
$$

Notice that this formula for $\mathbf{F}_{\mathbf{w}}$ mirrors the PST procedure, justifying our use of the $\mathcal{D}\mathbf{w}$ notation.

So $\mathbf{F}_{\mathbf{w}}$ is simply the uncentered covariance of the log-likelihood gradients, which resembles one of our formulas for the Fisher information matrix over probability distributions (Eqn. 8). Hence, we normally just refer to $\mathbf{F}$ as the **Fisher information matrix** for the network, rather than interpreting it as a pullback metric. However, it's important to remember that the original definition of the Fisher information matrix (for probability distributions) doesn't directly apply to neural nets, and we must instead construct $\mathbf{F}_{\mathbf{w}}$ as a pullback metric.

To apply our PST sampling function to the Fisher information matrix, we simply need to provide a routine that samples vectors with mean $\mathbf{0}$ and covariance $\mathbf{F}_{\mathbf{z}}$. We can do this by sampling the targets from the model's output distribution and computing the output space gradient (see Eqn. 8). Here is an implementation of the most common use cases: Bernoulli and categorical distributions.

```
def bernoulli_fisher_metric_sample(logits, rng):
    y = nn.sigmoid(logits)
    t = random.bernoulli(rng, y)
    return y - t

def categorical_fisher_metric_sample(logits, rng):
    y = nn.softmax(logits)
    t_idxs = random.categorical(rng, logits)
    t = nn.one_hot(t_idxs, logits.shape[-1])
    return y - t
```

Note that the Fisher information matrix shouldn't *automatically* be preferred over pullbacks of other output space metrics such as Euclidean distance. For classification problems, one can make arguments in favor of Euclidean distance on the logits. There have not yet been any direct comparisons of the two metrics. Since changing from one to the other requires only a few lines of code, it can be worth trying both for any particular problem.

When using Fisher information, there's an important gotcha: $\mathbf{F}_{\mathbf{w}}$ is the gradient covariance *when the targets are sampled from the model's predictions*. In particular, it is not the same as the covariance of the training gradients! That matrix is known as the **empirical Fisher matrix**, and

is used in some optimization algorithms such as Adagrad, RMSprop, and Adam:

$$\mathbf{F}_{\text{emp}} = \mathbb{E}_{(\mathbf{x},\mathbf{t})\sim p_{\text{data}}}[\nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w}) \nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})^\top] \tag{33}$$

The empirical Fisher matrix $\mathbf{F}_{\text{emp}}$ can behave very differently from the true Fisher. In particular, unlike the true Fisher, $\mathbf{F}_{\text{emp}}$ cannot be interpreted as an approximation to the Hessian. Yet, many papers simply substitute $\mathbf{F}_{\text{emp}}$ for $\mathbf{F}$ without noting the distinction. See Kunstner et al. (2019) for more in-depth discussion of this point.

For completeness, we also introduce the **gradient covariance matrix**, which will become important in Chapter 7, when we discuss stochastic optimization:

$$\mathbf{C} = \text{Cov}_{(\mathbf{x},\mathbf{t})\sim p_{\text{data}}}(\nabla \mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})). \tag{34}$$

The matrices $\mathbf{F}_{\text{emp}}$ and $\mathbf{C}$ are closely related, but not identical because the training gradients are not zero mean (except at a stationary point). We derive the relationship between these matrices using the identity $\mathbb{E}[\mathbf{v}\mathbf{v}^\top] = \text{Cov}(\mathbf{v}) + \mathbb{E}[\mathbf{v}]\mathbb{E}[\mathbf{v}]^\top$:

$$\mathbf{F}_{\text{emp}} = \mathbf{C} + \nabla \mathcal{J}(\mathbf{w}) \nabla \mathcal{J}(\mathbf{w})^\top, \tag{35}$$

where the gradient in the second term denotes the full-batch gradient.

Figure 5 summarizes the matrices we've introduced so far. To avoid information overload, $\mathbf{F}_{\text{emp}}$ and $\mathbf{C}$ are still grayed out, since we'll cover them in more detail in later lectures.

6 Invariance

Various fields have bookkeeping devices which prevent us from performing nonsensical operations. In the sciences, we assign units to all our quantities so that we don't accidentally add feet and miles. Similarly, most programming languages have some sort of type system which determines what sorts of operations we can perform on what sorts of data. Hence, if we want to add an int and a float, we need to first cast the int into a float, rather than simply interpreting its bit representation as an integer. In many cases, these bookkeeping devices can provide significant hints about how to solve a problem. E.g., physicists can often correctly guess a formula just by making sure the units match, and users of a programming language with a sophisticated type system (e.g. Haskell) will attest that getting one's program to compile often clears up a lot of conceptual misunderstandings.

Gradient descent is making the mathematical equivalent of a type error. Natural gradient is what you get when you correctly typecast the Euclidean gradient.

To see why this is, let's talk about invariances. Suppose we are fitting a linear regression model

$$y = w_1 x_1 + w_2 x_2 + b,$$

where x_1 is an input feature representing time, measured in minutes, x_2 is an input feature representing length in feet, and y is the output representing money, measured in dollars. We can determine the dimensions of the weights to make everything consistent:

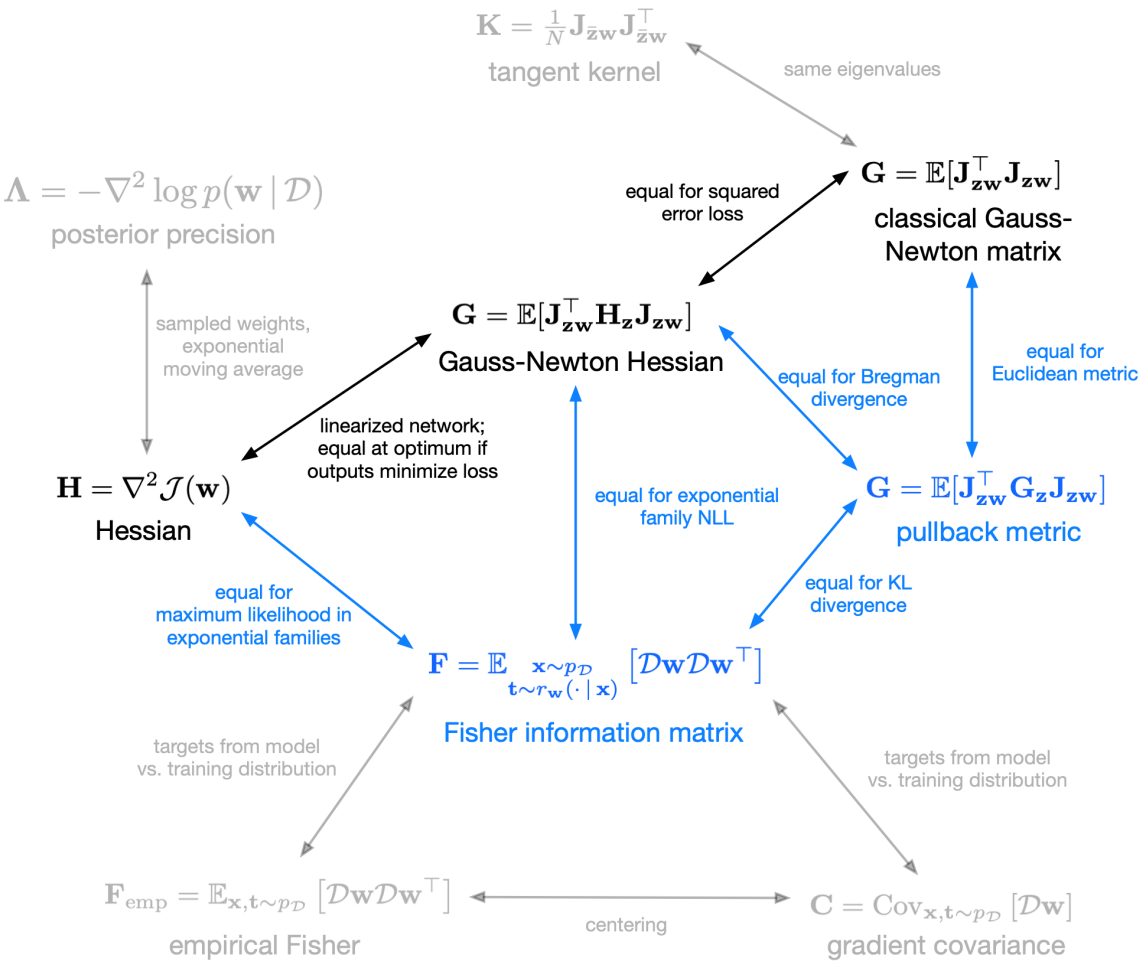
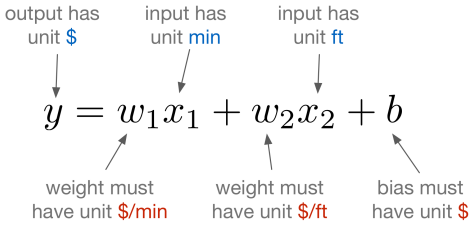
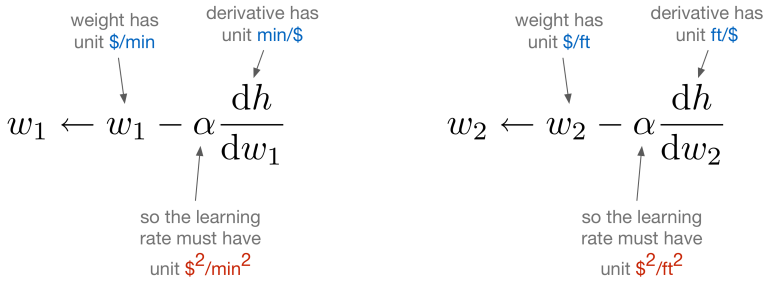


Figure 5: A summary of the relationships between the matrices used in this course. New items are in blue, and items yet to be covered are grayed out.

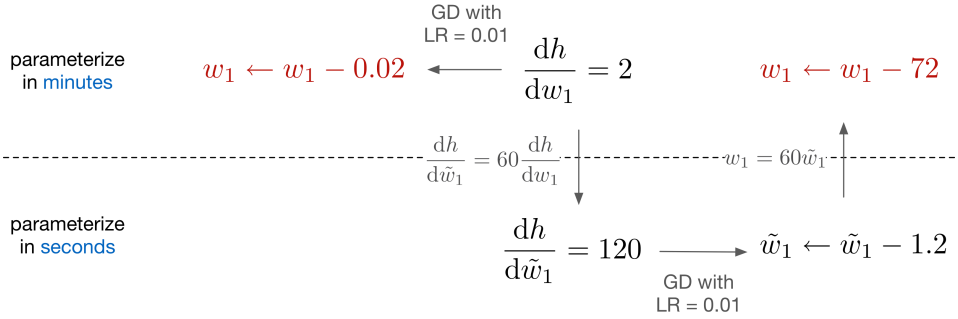


For the derivatives $\mathrm{d}h/\mathrm{d}w_i$, the weight appears in the denominator, so the units must be min/\$ and ft/\$, respectively. Now let's attempt to attach dimensions to the gradient descent update:



So the two occurrences of the global learning rate require two different units. There's no unit we can assign it to make everything dimensionally consistent!

How does the dimensional inconsistency affect optimization? Suppose we change the format of the data so that the first input is represented in seconds, i.e. $\tilde{x}_1 = 60x_1$. It's pretty easy to modify our regression weights to fit the new dataset; we simply take $\tilde{w}_1 = w_1/60$. This new model will make exactly the same predictions, since $\tilde{w}_1\tilde{x}_1 + w_2x_2 + b = w_1x_1 + w_2x_2 + b$. Clearly, the regression problem is essentially the same whether the inputs are in seconds or minutes, so one would hope an algorithm would behave equally well in either representation. Unfortunately, this is not true for gradient descent. Suppose we see a training example for which $\frac{\partial h}{\partial w_1} = 2$, so that gradient descent with a learning rate of 0.01 gives $w_1 \leftarrow w_1 - 0.02$. Look what happens when we do gradient descent in both coordinate systems:



So as a result of changing from minutes to seconds, the effective learning rate got 3600 times larger!

The property which failed to hold just now is known as **invariance**. We had two equivalent ways of representing the problem – seconds or minutes – and we'd like our algorithm to behave the same regardless of the fairly arbitrary choice of representation. In particular, the sequence of iterates

Sometimes this property is referred to as **covariance**. Here, the idea is that if you stretch out the coordinate system, the update should stretch in the same way.

should be functionally equivalent, i.e. make the same predictions given the same data.

Mathematicians confront this sort of problem quite regularly: they'd like to be sure that some property of a mathematical object doesn't depend on how that object is represented. They general strategy they adopt is to define more abstract notions such as vector spaces or manifolds without any privileged coordinate system. If one can build something out of these more abstract objects, then one achieves invariance for free.

6.1 Of Music and Manifolds

Natural gradient comes from the field of differential geometry. The study of differential geometry applied to manifolds (spaces) of probability distributions is known as information geometry, and the classic reference text is Amari and Nagaoka (2000)'s book *Methods of Information Geometry*. Ollivier (2015) gives a nice discussion of practical neural net training based on differential geometry. Proper coverage of differential geometry is well beyond the scope of this course, but here is just a short teaser.

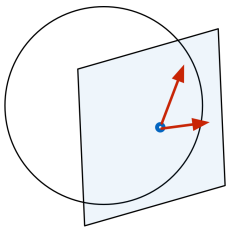
Here are some of the basic mathematical objects we make use of:

Manifolds are spaces which can be given local coordinate systems, the way a section of the Earth's surface can be represented with a map. A sphere is, in fact, a good example of a manifold. But unlike the sphere, not all manifolds are embedded in some higher dimensional Euclidean space; for instance, the most important examples of manifolds in information geometry are families of probability distributions. The study of manifolds without reference to an embedding space is known as Riemannian geometry.

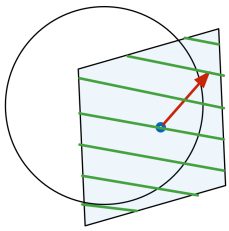
Tangent vectors represent instantaneous velocities of particles moving along the manifold. The space of tangent vectors at a given point on the manifold (called the tangent space) is a vector space, so we can do things like take linear combinations of tangent vectors.

Cotangent vectors are linear functions on the tangent space. E.g., the coefficient in the first-order Taylor approximation (the thing we normally call the "gradient") is really a cotangent vector. Like vectors, covectors can be represented in a coordinate system as a list of numbers. The difference is how those numbers change as we change the coordinate system.

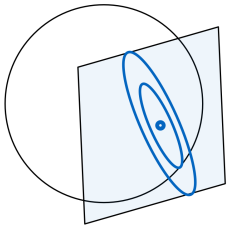
Riemannian metrics are inner products on the tangent space. The inner product of a vector with itself defines the "squared length" of that vector.



Two vectors in the tangent space to M.



A vector in the tangent space, and a covector, visualized as a linear functional. When we apply this covector to this vector, it evaluates to 2 (count the contours).



A Riemmanian metric gives an inner product on the tangent space at every point. This induces a norm.

Since we generally identify vectors and covectors in Euclidean spaces, it may seem surprising that we need to distinguish them. But they are in fact distinct, and one important difference between them is the set of operations we're allowed to perform on them. Suppose we have a map f from a manifold $\mathcal{M}$ to another manifold $\mathcal{N}$. Suppose we have a point x on $\mathcal{M}$. We can **push it forward** to $\mathcal{N}$ by computing its image, i.e. $f_\star x = f(x)$. On the other hand, if we have a function g on $\mathcal{N}$, we can **pull it back** to $\mathcal{M}$ by defining a function $(f^\star g)(x) = g(f(x))$. Observe that we can't do this the other way around. I.e., there's no way to pull back points, since if f isn't invertible, there may be no unique point $x \in \mathcal{M}$ which maps to a given $y \in \mathcal{N}$. For the same reason, there's no way to push forward functions.

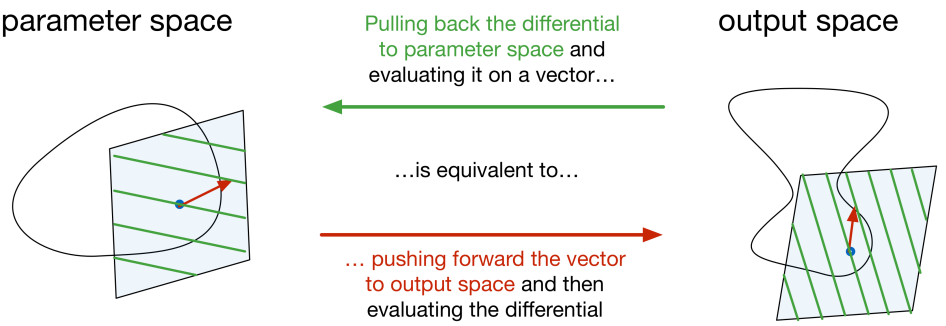
Whether objects can be pushed forward or pulled back is a fundamental distinction. Here are some things that can be pushed forward:

- points
- tangent vectors
- probability distributions

And some things that can be pulled back:

- functions
- covector fields
- Riemannian metrics

Many of our operations on neural nets can be described in the language of differential geometry. When we do backprop, we first compute the loss derivatives on the output space; this is really a covector called the **differential**. It's a covector because it's a linear function that tells us, for a given change to the outputs, approximately how much will the loss change? Backprop then pulls this differential back to parameter space. We now have a covector on parameter space which tells us, for a given change to the parameters, approximately how much the loss will change. Here it is visually:



But there's a problem. In order to update the parameters, we need a rate of change, and rates of change are represented by vectors. There's no fully general way to convert a covector into a vector. But we can do such a conversion if we're fortunate enough to have a Riemannian metric on parameter space. This is because a Riemannian metric defines an isomorphism between covectors and vectors, called the **musical isomorphism** because it's often notated using sharps and flats.

To see how the musical isomorphism works, forget manifolds for a moment, and just think about vector spaces. Suppose we have a vector space $\mathcal{V}$ with an inner product $\langle \cdot, \cdot \rangle$. If we're given a vector v, consider the function $v^\flat(u) = \langle v, u \rangle$. (The superscript is the musical flat, which gives the musical isomorphism its name.) This function $v^\flat$ is a linear function of u, which is the same thing as a covector. This mapping from vectors to covectors is linear:

$$(\alpha v_1 + \beta v_2)^\flat = \langle \alpha v_1 + \beta v_2, \cdot \rangle = \alpha \langle v_1, \cdot \rangle + \beta \langle v_2, \cdot \rangle = \alpha v_1^\flat + \beta v_2^\flat.$$

You can also show this mapping is invertible, i.e., if you're given a covector ω – remember, this is just a linear function on the vector space $\mathcal{V}$ – you can find a unique vector (denoted, naturally, $\omega^\sharp$) such that $\omega(u) = \langle \omega^\sharp, u \rangle$ for all u. Hence, this $\sharp/\flat$ mapping is an isomorphism – the musical isomorphsim.

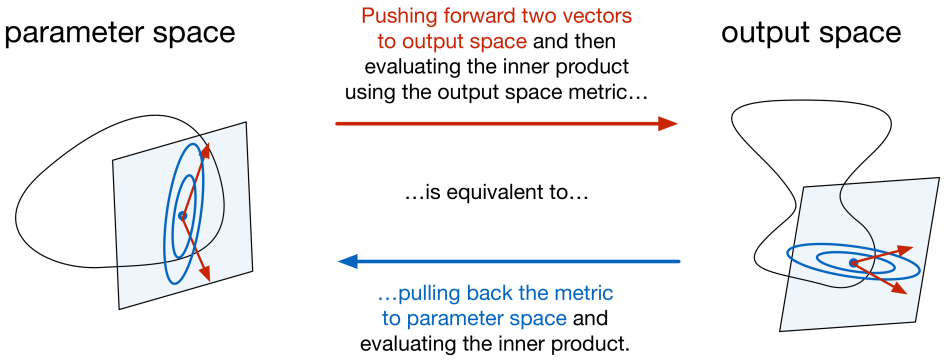
Now back to manifolds. Remember that we have a covector (the differential) and we need a vector. A Riemannian metric is just an inner product on the tangent space for every point on a manifold. Hence, we can use it to convert between vectors and covectors using the musical isomorphism. When we use the Riemannian metric to convert the differential into a vector, we get the **gradient**, i.e. the direction of fastest increase in the cost function with respect to that metric.

How do we get a metric on parameter space? One option would be the Euclidean metric, i.e. the ordinary Euclidean inner product. Applying this metric would result in the ordinary Euclidean gradient descent update. But this metric wouldn't be **natural**, because it's defined in terms of an arbitrary coordinate system. If we reparameterized our neural net to use tanh activation functions instead of logistic, we'd get a different metric.

I guess a musical note is natural if it's not sharp or flat?

Here's how we can construct a natural metric. In many applications of neural nets, the outputs of the network represent probability distributions. Luckily, there's a natural metric for manifolds of probability distributions: the Fisher metric. (Recall that the Fisher metric is the second-order Taylor approximation to KL divergence, which is intrinsic to the distributions and not defined in terms of parameters.) To get a metric on parameter space,

we just pull back the Fisher metric from output space to parameter space. How do we pull back a metric? There's quite an elegant definition: first note that we can push vectors forward from parameter space to output space by computing the change to z that results from a small change to θ. To evaluate the pullback metric on two parameter space vectors, we simply push those vectors forward to output space and then evaluate the output space metric.



Putting this all together: backpropagation takes the differential on output space and pulls it back to parameter space. We define a natural Riemannian metric on output space (such as the Fisher metric) and pull it back to parameter space. We use the musical isomorphism for this metric to convert the differential into a vector, which we call the natural gradient. All of these operations are intrinsic to the manifolds, so the resulting update direction is mathematically guaranteed to be dimensionally correct, and invariant to smooth reparameterization (though the iterates of a discretized algorithm, which takes discrete steps in these update directions, won't enjoy quite as many invariances).

References

Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10:251–276, 1998.

Shun-ichi Amari. *Information Geometry and its Applications*. Springer, 2016.

Shun-ichi Amari and Hiroshi Nagaoka. *Methods of Information Geometry*. AMS and Oxford University Press, 2000.

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

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

Yann Ollivier. Online natural gradient as a Kalman filter. *Electronic Journal of Statistics*, 12:2930–2961, 2018.