Chapter 2: Taylor Approximations

Roger Grosse

1 Introduction

One of the most powerful tools for understanding a nonlinear system is to linearize that system around some point of interest (such as a stationary point). For one thing, it's hard to exactly analyze the dynamics of most nonlinear systems. But if we take a first-order Taylor approximation to the dynamics, then we get a linear system, which can be analyzed exactly. Think back to classical mechanics: if you take a first-order Taylor approximation to the potential energy of a system around a stable equilibrium point, you arrive at a system known as a simple harmonic oscillator — one of the easiest physical systems to analyze.

This lecture is all about various kinds of Taylor approximations for neural networks. We'll start with first-order Taylor approximations. This includes the example that most readers are already familiar with: the gradient of a cost function, which can be computed with backpropagation, also known as reverse mode automatic differentiation. A possibly less familiar example is directional derivatives, which let us approximate how a network's outputs will change if the weights or the activations are perturbed by a small amount. Directional derivatives can be efficiently computed with forward mode automatic differentiation. Both the gradient and directional derivatives can be viewed in terms of computing with the network's Jacobian, a perspective that lets us use automatic differentiation as a foundation for second-order Taylor approximations.

Naturally, once we're done with first-order Taylor approximations, we'll turn our attention to second-order Taylor approximations. There are different quantities we may want to approximate, and they give us different but related matrices.

The most well-known second-order Taylor approximation is the Hessian, or the second derivatives of the cost function with respect to the weights of the network. By taking the second-order Taylor approximation to the cost function around its optimum, we arrive at a convex quadratic objective, thus reducing the system to the one we analyzed in Lecture 1. This gives a convenient way to understand the behavior of gradient descent.

It's common to approximate neural net Hessians by linearizing the network. This results in an approximate Hessian matrix called the Gauss-Newton Hessian. While this matrix is most known for its role in second-order optimization, we'll start with a different use case: influence functions, which tell us how our model's outputs would change if we slightly perturbed the training data.

The other thing we'd like to approximate is the amount by which a network's outputs change if the weights are adjusted slightly. Taking the

second-order Taylor approximation gives a class of matrices I'll term pullback metrics, the most famous of which is the Fisher information matrix. Pullback metrics and Fisher information will be covered in Lecture 3.

The reason the ideas of this lecture are not part of the basic neural net pedagogy is likely that the main deep learning software frameworks don't make it very convenient to use these tools. While TensorFlow and PyTorch support automatic gradient computation as one of their main features, directional derivatives and second derivatives are much more cumbersome. Therefore, for this lecture, and throughout the course, we'll be using JAX, an elegant autodiff framework which supports all of this lecture's mathematical tools in only a few lines of code, with the efficiency we'd expect of a modern deep learning framework. JAX might just take over in a few years time, so you should get a head start and learn it now!

2 Preliminaries

For most of this course, we'll assume a supervised learning setting, although most of the ideas apply just as well to other settings, such as generative modeling or reinforcement learning. Assume the **inputs** $\mathbf{x}$ and **labels** $\mathbf{t}$ are jointly drawn from the **data generating distribution**, i.e. $(\mathbf{x}, \mathbf{t}) \sim p_{\text{data}}$. We are given a finite **training set** $\{(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})\}_{i=1}^N$. There is a **loss function** $\mathcal{L}(\mathbf{y}, \mathbf{t})$ which determines how unhappy we are when we predict $\mathbf{y}$ and the true label is $\mathbf{t}$. We make predictions using a network architecture parameterized by $\mathbf{w}$ which computes a function $f(\mathbf{w}, \mathbf{x})$. We are interested in learning $\mathbf{w}$ which achieve small **risk**, or **generalization loss**,

$$\mathcal{R}(\mathbf{w}) = \mathbb{E}_{(\mathbf{x},\mathbf{t})\sim p_{\text{data}}}[\mathcal{L}(f(\mathbf{w}, \mathbf{x}), \mathbf{t})]. \tag{1}$$

We do this by minimizing a **cost function** corresponding to the **empirical risk** over the training set:

$$\begin{aligned} \mathcal{J}(\mathbf{w}) &= \frac{1}{N}\sum_{i=1}^N \mathcal{J}^{(i)}(\mathbf{w}) \\ &= \frac{1}{N}\sum_{i=1}^N \mathcal{J}_{\mathbf{x}^{(i)},\mathbf{t}^{(i)}}(\mathbf{w}) \\ &= \frac{1}{N}\sum_{i=1}^N \mathcal{L}(f(\mathbf{w}, \mathbf{x}^{(i)}), \mathbf{t}^{(i)}). \end{aligned} \tag{2}$$

Here, $\mathcal{J}^{(i)}(\mathbf{w})$ is a shorthand for the loss on the ith training example, and $\mathcal{J}_{\mathbf{x},\mathbf{t}}(\mathbf{w})$ is a shorthand for the loss on input $\mathbf{x}$ and label $\mathbf{t}$. The average over training examples can also be interpreted as the expectation under the **empirical distribution**, which puts a point mass of $1/N$ on each training example.

A symmetric matrix $\mathbf{A}$ is **positive definite** if $\mathbf{v}^\top \mathbf{A}\mathbf{v} > 0$ for all $\mathbf{v} \neq \mathbf{0}$, and it is **positive semidefinite (PSD)** if $\mathbf{v}^\top \mathbf{A}\mathbf{v} \geq 0$ for all $\mathbf{v}$. It can be shown that $\mathbf{A}$ is positive definite iff all of its eigenvalues are positive, and PSD iff all of its eigenvalues are nonnegative. We use the following partial order over symmetric matrices: $\mathbf{A} \succ \mathbf{B}$ if $\mathbf{v}^\top \mathbf{A}\mathbf{v} > \mathbf{v}^\top \mathbf{B}\mathbf{v}$ for any $\mathbf{v} \neq \mathbf{0}$, or equivalently, if $\mathbf{A} - \mathbf{B}$ is positive definite. The relations $\succeq$, $\prec$, etc. are defined analogously.

3 First-Order Taylor Approximations

Let $\mathbf{y} = f(\mathbf{w})$ be a vector-valued function of a vector $\mathbf{w}$ which is differentiable at a point $\mathbf{w}_0$. (The case where f is scalar-valued or $\mathbf{w}$ is a scalar can be handled with a 1-dimensional vector.) **Taylor's Theorem** implies that f can be approximated around $\mathbf{w}_0$ as follows:

$$f(\mathbf{w}) = f(\mathbf{w}_0) + \mathbf{J}_{\mathbf{yw}}(\mathbf{w} - \mathbf{w}_0) + o(\|\mathbf{w} - \mathbf{w}_0\|), \tag{3}$$

where $\mathbf{J}_{\mathbf{yw}}$ is the **Jacobian matrix** (or just **Jacobian**) whose entries are the partial derivatives:

$$[\mathbf{J}_{\mathbf{yw}}]_{ij} = \frac{\partial y_i}{\partial w_j}. \tag{4}$$

and the little-o notation implies that the remainder goes to 0 faster than $\|\mathbf{w} - \mathbf{w}_0\|$ as $\mathbf{w} \to \mathbf{w}_0$. This approximation is called the **first-order Taylor approximation**, or **linearization**, of f. The general first-order Taylor approximation is a bit abstract, so let's start with some special cases.

In deep learning, we deal with non-differentiable functions all the time, such as the ubiquitous ReLU activation function. As long as we're not at a non-differentiable point, Taylor's Theorem holds. Even if we're at the non-differentiable point, we can usually just ignore the problem and compute derivatives anyway. The Taylor approximation won't be accurate, but often that doesn't matter.

3.1 The Gradient Vector

The most well-known example of a first-order Taylor approximation, and one which we take for granted in deep learning, is the **gradient** $\nabla\mathcal{J}(\mathbf{w})$ of a cost function $\mathcal{J}$:

$$[\nabla\mathcal{J}(\mathbf{w}_0)]_j = \frac{\partial\mathcal{J}}{\partial w_j}\Big|_{\mathbf{w}=\mathbf{w}_0} \tag{5}$$

This is just the special case of Eqn. 3 where f is scalar-valued:

$$\mathcal{J}(\mathbf{w}) = \mathcal{J}(\mathbf{w}_0) + \nabla\mathcal{J}(\mathbf{w}_0)^\top(\mathbf{w} - \mathbf{w}_0) + o(\|\mathbf{w} - \mathbf{w}_0\|). \tag{6}$$

Mathematically, $\nabla\mathcal{J}(\mathbf{w}_0)$ should be a row vector (i.e. $1 \times N$ matrix), but in machine learning we usually treat it as a column vector. That's the convention we'll adopt in this course.

By linearity of derivatives, the **full-batch gradient** decomposes as an average of the gradients for individual training examples:

$$\nabla\mathcal{J}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\nabla\mathcal{J}^{(i)}(\mathbf{w}). \tag{7}$$

The gradient is used in nearly every optimization algorithm for neural networks, most simply gradient descent:

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \alpha\nabla\mathcal{J}(\mathbf{w}^{(k)}). \tag{8}$$

In practice, computing gradients on the entire dataset is prohibitive, so instead we use **stochastic gradient descent**, where the gradients are estimated from smaller batches of data.

As the reader is probably aware, the gradient can be computed using **backpropagation**, or **reverse mode autodiff**. See the CSC413 course notes for a detailed explanation of backprop and how it's implemented in autodiff software frameworks. In JAX, gradients are computed using the `grad` function. Actually, `grad` is just a thin wrapper around a more general function, `vjp` (short for **vector-Jacobian product**) whose job it is to compute $\mathbf{J}^\top\mathbf{v}$, where $\mathbf{J}$ is the Jacobian of a function f at a point $\mathbf{x}$, and $\mathbf{v}$ is a vector. Specifically, `make_vjp` takes in a function and its arguments, and

returns a function that takes a vector $\mathbf{v}$ and computes $\mathbf{J}^\top \mathbf{v}$. Observe that the Jacobian of a scalar-valued function $\mathcal{J}$ at $\mathbf{w}$ is simply $\nabla \mathcal{J}(\mathbf{w})^\top$, i.e. the gradient viewed as a row vector. Hence, computing $\nabla \mathcal{J}(\mathbf{w})$ is equivalent to computing $\mathbf{J}^\top \mathbf{e}$, where $\mathbf{e}$ is a 1-dimensional vector whose single entry is 1. This can be implemented as follows:

This code is simplified in that it doesn't support the full range of use cases (multiple arguments, vector-valued outputs, etc.), but otherwise it's pretty faithful to the true implementation.

```
def my_grad(f):
    def grad_f(w):
        ans, f_vjp = vjp(f, w)
        return f_vjp(1.)[0]
    return grad_f
```

3.2 Directional Derivatives

Suppose we make a small perturbation to the weights, $\mathbf{w} = \mathbf{w}_0 + \Delta\mathbf{w}$. This will cause a small perturbation to the outputs, $\mathbf{y} = \mathbf{y}_0 + \Delta\mathbf{y}$, where according to Eqn. 3,

$$\Delta\mathbf{y} = \mathcal{R}_{\Delta\mathbf{w}} f(\mathbf{w}) + o(\|\Delta\mathbf{w}\|), \tag{9}$$

where

$$\mathcal{R}_{\mathbf{\Delta w}} f(\mathbf{w}) = \lim_{h \to 0} \frac{f(\mathbf{w} + \Delta\mathbf{w}) - f(\mathbf{w})}{h} = \mathbf{J}_{\mathbf{yw}} \Delta\mathbf{w}$$

is the **directional derivative** of f in the direction $\Delta\mathbf{w}$. This is sometimes also referred to as the **Gateaux derivative**, or **R-operator (R-op)**. Visually, if f represents a curve or surface in $\mathbb{R}^n$, then the directional derivatives represent tangent vectors to the curve or surface.

JAX provides the `jacfwd` function for computing directional derivatives. Behind the scenes, directional derivatives are computed using a procedure closely analogous to backprop (reverse mode autodiff), known as **forward mode autodiff**. This procedure is actually a bit simpler than reverse mode autodiff, because it works forwards through the computation graph rather than backwards.

Since forward mode works forward through the computation graph, it is also more memory efficient than reverse mode. In the most straightforward implementation of reverse mode, we need to store the value for every node in the graph. For forward mode, we are free to forget a node once the values and derivatives for its children have been computed.

From this description, it would seem like an implementation of forward mode is analogous to one of reverse mode: it would require a simple traversal of the graph, combined with **Jacobian-vector products (JVPs)** for each of the elementary operations. Indeed, it can be implemented this way. But amazingly, if one has already implemented reverse mode autodiff, it is possible to implement forward mode in perhaps the most elegant 3 lines of code I've ever seen:

```
def my_jvp(f, w, R_w):
    ans, f_vjp = vjp(f, w)
    _, f_vjp_vjp = vjp(f_vjp, np.zeros_like(ans))
    return f_vjp_vjp((R_w,))[0]
```

This clever implementation of forward mode is due to Jamie Townsend. `https://github.com/HIPS/autograd/pull/175#issuecomment-306524258`

Let's unpack this. As discussed above, `vjp` is the reverse mode autodiff function which `grad` wraps around, and it returns a function that takes in a vector $\mathbf{v}$ and computes $\mathbf{J}^\top \mathbf{v}$. This is a linear function (in $\mathbf{v}$), so its Jacobian is $\mathbf{J}^\top$. Hence, calling reverse mode autodiff on *it* is equivalent to multiplying by $(\mathbf{J}^\top)^\top = \mathbf{J}$. This is what the second line does: it produces a function that takes in a vector $\mathbf{v}$ and computes $\mathbf{Jv}$. The third line calls this function on `R_w`, thereby computing the directional derivative. Beautiful!

3.3 Computing with the Jacobian

The Jacobian of a function (such as a neural network) is a very useful conceptual tool, since it unifies many of the operations we commonly want to compute for a neural network. As we saw, both gradients and directional derivatives can be viewed as multiplying by the Jacobian or its transpose.

I thought maybe JAX was short for "Jacobians", but the creators tell me this isn't the case.

JAX provides the `jacfwd` and `jacrev` routines for computing the full Jacobian matrix, if you really want to. Note, however, that the Jacobian can be a very large matrix: it is $M \times N$, where M and N are the output and input dimensions, respectively. But suppose the domain of f is the weights of a classification neural network and the output is the vector of logits for a training example. Then N is the number of parameters (in the millions for a modern ImageNet classifier), and M is the number of categories (1000 for ImageNet). Hence, the Jacobian has billions of entries, so it's prohibitive to compute and to store.

Fortunately, there's rarely a reason to compute $\mathbf{J_{yw}}$ explicitly. Instead, we find some way to write our algorithm in terms of products of $\mathbf{J_{yw}}$ or $\mathbf{J_{yw}^\top}$ with vectors. Multiplying $\mathbf{J_{yw}}$ by a vector (called a **Jacobian-vector product**, or **JVP**) is done with forward mode autodiff, and multiplying $\mathbf{J_{yw}^\top}$ by a vector (called a **vector-Jacobian product**, or **VJP**) is done with reverse mode autodiff. This is an instance of a more general algorithmic pattern called **implicit matrix-vector products**, and we'll see more examples of this later in the lecture. Both JVPs and VJPs are linear time operations, in that the number of computations is a small multiple of the cost of evaluating $f(\mathbf{w})$. Therefore, as long as we can phrase an iterative update rule in terms of VJPs and JVPs, its running time will be on the same order as backprop.

The fact that VJPs and JVPs are both linear time is kind of an amazing fact, and leads to lots of useful algorithms.

4 The Hessian Matrix

To understand the behavior of gradient-based optimization algorithms within a "small" region of weight space, it's natural to take a second-order Taylor approximation to the cost function. To do this, we require the **Hessian matrix** at $\mathbf{w}$, denoted as $\mathbf{H}$, whose entries are the second derivatives of the cost function:

We'll come back later to the question of just how "small" this region needs to be.

$$\begin{aligned}\mathbf{H} &= \nabla^2 \mathcal{J}(\mathbf{w}_0) \\ &= \frac{1}{N}\sum_{i=1}^{N} \nabla^2 \mathcal{J}^{(i)}(\mathbf{w}_0) \\ H_{ij} &= \frac{\partial^2 \mathcal{J}}{\partial w_i w_j}\Big|_{\mathbf{w}=\mathbf{w}_0}\end{aligned}$$

Note that $\mathbf{H}$ is symmetric because the mixed partial derivatives are equal, i.e. $\frac{\partial^2 \mathcal{J}}{\partial w_i \partial w_j} = \frac{\partial^2 \mathcal{J}}{\partial w_j \partial w_i}$ for each (i, j). Assuming $\mathcal{J}$ is twice differentiable, this gives a second-order Taylor approximation to the cost function around a point $\mathbf{w}_0$:

$$\mathcal{J}(\mathbf{w}) = \mathcal{J}(\mathbf{w}_0) + \nabla \mathcal{J}(\mathbf{w}_0)^\top (\mathbf{w} - \mathbf{w}_0) + \tfrac{1}{2}(\mathbf{w} - \mathbf{w}_0)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_0) + o(\|\mathbf{w} - \mathbf{w}_0\|^2). \tag{10}$$

The Hessian matrix measures the **curvature** of $\mathcal{J}$ at a point $\mathbf{w}$. If $\mathbf{v}^\top \mathbf{H}\mathbf{v} > 0$ for some direction $\mathbf{v}$, that means the function curves upwards if you move along the direction $\mathbf{v}$. Likewise, if $\mathbf{v}^\top \mathbf{H}\mathbf{v} < 0$, it curves downwards. If $\mathbf{v}^\top \mathbf{H}\mathbf{v} = 0$, then the second-order curvature is 0, and the behavior will depend on the higher-order derivatives. Let's see what the Hessian tells us about a function.

The quantity $\frac{\mathbf{v}^\top \mathbf{H}\mathbf{v}}{\mathbf{v}^\top \mathbf{v}}$, called the **Rayleigh quotient**, measures *how fast* the function curves up or down if you move along the direction $\mathbf{v}$.

An important class of functions in optimization is **convex functions**, which are (roughly speaking) "bowl-shaped". Mathematically, a function $\mathcal{J}$ is convex if and only if the line segment connecting any two points on its graph lies entirely above the graph:

$$\mathcal{J}(\lambda \mathbf{w}_1 + (1-\lambda)\mathbf{w}_0) \leq \lambda \mathcal{J}(\mathbf{w}_1) + (1-\lambda)\mathcal{J}(\mathbf{w}_0) \qquad \text{for any } \mathbf{w}_0, \mathbf{w}_1, \text{ and } 0 \leq \lambda \leq 1. \tag{11}$$

If the Hessian exists, then there's another equivalent characterization of convex functions: the Hessian is always PSD. (If $\mathbf{H}$ is always positive *definite*, then $\mathcal{J}$ is **strictly convex**.) This characterization essentially says that a function is convex iff it curves upwards in every direction.

See Section 2 for definitions of *positive semidefinite (PSD)* and related terms.

Convex functions are very convenient to minimize, because there are no **spurious local optima**, i.e. all local optima are also global optima. Unfortunately, most cost functions for training neural nets are non-convex. For non-convex cost functions, there are many different kinds of **stationary points**, i.e. points $\mathbf{w}_\star$ such that $\nabla \mathcal{J}(\mathbf{w}_\star) = \mathbf{0}$. The eigenvalues of $\mathbf{H}$ can help us categorize the stationary points:

Most neural net training is inevitably non-convex because the networks satisfy certain symmetries which, in combination with Eqn. 11, would imply that a certain trivial solution is optimal. For more details, see the "Optimization" section of the CSC413 lecture notes.

- If $\mathbf{H}$ is positive definite (or equivalently, all of its eigenvalues are positive), then $\mathbf{w}_\star$ is a local optimum. (Intuitively, the function curves upwards, so walking in any direction will take you uphill.)
- If $\mathbf{H}$ is negative definite (or equivalently, has all negative eigenvalues), then $\mathbf{w}_\star$ is a local maximum. (This is unusual in neural net training.)
- If $\mathbf{H}$ has some positive and some negative eigenvalues, then $\mathbf{w}_\star$ is a **saddle point**, where some directions curve upwards and other directions curve downwards. (Saddle points are much more common.)
- If $\mathbf{H}$ is positive semidefinite, but some of the eigenvalues are 0, then we can't say whether it's a local optimum. The shape of the function depends on the higher order derivatives.

Consider, e.g., the univariate function $f(w) = -w^4$ at $w = 0$.

4.1 Gradient Descent Dynamics

The convenient thing about second order Taylor approximations is that we can analyze optimization dynamics in closed form. While gradient descent on general functions is a complicated nonlinear system, once we take the second-order Taylor approximation around a stationary point $\mathbf{w}_\star$, we've reduced the problem to the same quadratic system we analyzed in last week's lecture. For simplicity, in this section we'll assume none of the eigenvalues of $\mathbf{H}$ are exactly 0. Recall that the gradient descent iterates for the approximate objective (Eqn. 10) are given by:

See Lecture 1 for the derivation.

$$\mathbf{w}^{(k)} = \mathbf{w}_\star + (\mathbf{I} - \alpha \mathbf{H})^k (\mathbf{w}^{(0)} - \mathbf{w}_\star).$$

If we rotate to a coordinate system whose axes are the eigenvectors of $\mathbf{H}$, then each coordinate evolves according to

$$\tilde{w}_i^{(k)} = \tilde{w}_{i\star} + (1 - \alpha\tilde{h}_i)^k(\tilde{w}_i^{(0)} - \tilde{w}_{i\star}),$$

where $\tilde{h}_i$ is the corresponding eigenvector. Recall that if $0 < \alpha\tilde{h}_i < 2$, then coordinate i converges exponentially, whereas if $\alpha\tilde{h}_i < 0$ or $\alpha\tilde{h}_i > 2$, then that coordinate diverges exponentially.

Since $\mathbf{H}$ is a symmetric matrix, the spectral decomposition from last lecture applies.

Recall that we are taking the Taylor approximation around a stationary point. First consider the case where this stationary point is a local optimum, and $\mathbf{H}$ is (strictly) positive definite. In this case, the approximate objective is a convex quadratic, just like we analyzed last week. As long as the learning rate is chosen such that $\alpha < \tilde{h}_{\max}^{-1}$, each coordinate converges exponentially, and directions of high curvature converge faster than directions of low curvature. Eventually, the loss will be dominated by the low curvature directions, and based on our analysis last week, the asymptotic convergence rate is determined by the condition number $\kappa = \tilde{h}_{\max}/\tilde{h}_{\min}$. Because $\mathbf{w}$ approaches $\mathbf{w}_\star$, the Taylor approximation remains accurate.

Notice I am not saying that the slow convergence along low curvature directions is *bad*. For any particular model, those directions might not be very important to optimize in, or they might even be harmful because they correspond to overfitting. (See, e.g., the discussion of whitening in Lecture 1.) Right now our focus is descriptive rather than prescriptive: we're trying to understand what is happening during training, and we'll figure out later what (if anything) to do about it.

If the stationary point $\mathbf{w}_\star$ isn't a local optimum, then it's probably a saddle point. Still assuming for simplicity that none of the eigenvalues of $\mathbf{H}$ are exactly 0, the only difference from the case of local optima is that now some of the eigenvalues are negative. In these coordinates, $1 - \alpha\tilde{h}_i > 1$, so as long as $\tilde{w}_i^{(0)} \neq \tilde{w}_i$, these coordinates move exponentially away from $\mathbf{w}_\star$. Of course, they don't move away forever; the weights will move far enough from the saddle point that the Taylor approximation is no longer accurate, and chances are, $\mathcal{J}$ will start curving upwards again. Gradient descent is said to have **escaped** the saddle point. In dynamical systems terminology, the difference between the behavior in these two cases is that local optima are **stable stationary points** and saddle points are **unstable stationary points**.

The one case where $\mathbf{w}$ might fail to escape a saddle point is if it happens that $\tilde{w}_i^{(0)} = \tilde{w}_i$ for some coordinate i. (As a special case, this happens if $\mathbf{w} = \mathbf{w}_\star$, i.e. we start out on the saddle point.) In this case, $\tilde{w}_i$ never changes, so the optimizer is stuck. Intuitively, it might seem unlikely to have $\tilde{w}_i^{(0)} = \tilde{w}_i$ exactly, and even if it does happen, one can probably escape in practice by slightly perturbing the updates. (I.e. the perturbation creates a small gap between $\tilde{w}_i$ and $\tilde{w}_{i\star}$, which then grows exponentially.) It's a bit delicate to actually prove this, but under some reasonable assumptions, this intuition does turn out to be mathematically correct. Gradient descent generally escapes saddle points efficiently, so we don't normally worry about them when training neural nets.

One way to get stuck at a saddle point is with a poor initialization, e.g. if all the weights are initialized to 0.

What does the eigenspectrum of $\mathbf{H}$ look like in practice? Much of the time, it will have many 0 eigenvalues, because the network is overparame-

Accurately estimating the eigenspectrum of $\mathbf{H}$ for a large modern neural net is a hard problem, perhaps even harder than training the network; see Adams et al. (2018); Ghorbani et al. (2019)

terized, and therefore some directions in weight space won't affect the loss. (More on this in a later lecture.) At initialization, $\mathbf{H}$ will generally have some positive and some negative eigenvalues of various magnitudes. However, the large negative eigenvalues are quickly eliminated because the optimizer tends to descend quickly in those directions. So in general, what we see during training is: some large positive eigenvalues, some small positive eigenvalues, some small (in magnitude) negative eigenvalues, and a great many 0 eigenvalues.

4.2 Computing with the Hessian

Analogously to the Jacobian, JAX provides the `hessian` function for computing the full Hessian, but you generally won't want to use this. The Hessian is a large matrix: its dimension is the number of parameters, which is typically in the millions. Therefore, it's impractical to compute and store it explicitly. Instead, we need to work with it indirectly using implicit matrix-vector products, analogously to how we handled $\mathbf{J}_{\mathbf{yw}}$ and $\mathbf{J}_{\mathbf{yw}}^\top$. There's a very elegant trick that lets us do this using automatic differentiation.

We can think of $\mathbf{H}$ as the Jacobian of the gradient. I.e., if we define the vector-valued function $g(\mathbf{w}) = \nabla \mathcal{J}(\mathbf{w})$, then $\mathbf{H}$ is the Jacobian of g. This can be translated directly into JAX code, by doing forward mode autodiff (using `jvp`) on the gradient. The following function computes the **Hessian-vector product** with a vector $\mathbf{v}$:

This code is taken from the JAX Autodiff Cookbook: `https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html`

```
def hvp(J, w, v):
    return jvp(grad(J), (w,), (v,))[1]
```

Notice that we're doing forward mode autodiff on a computation graph that is the result of reverse mode autodiff; hence, this is a strategy known as **forward-over-reverse**. Since both forms of autodiff are linear time operations, Hessian-vector products are linear time as well.

There are various other ways to compute Hessian-vector products with autodiff, all of which are also linear time.

4.3 Example: Weak Symmetry Breaking in Regularized Linear Autoencoders

From the preceding discussion, it might sound like the Taylor approximation to the cost function is only accurate once the weights are very close to a local optimum, i.e. the optimization is nearly done. On the contrary, the above analysis can often give us a lot of insight into the optimization dynamics throughout training. One reason for this is that neural nets can often learn interesting functions without the weights moving very far; hence, the Taylor approximation is accurate. We'll come back to this point in a later lecture. But right now, we'll look at an interesting example that is decidedly nonlinear, but where the Hessian nevertheless provides a lot of insight into the dynamics. This example is based on Bao et al. (2020).

A surprisingly fruitful source of insight into neural network dynamics is **linear networks**, i.e. networks whose activation function is the identity function. Such networks generally aren't practically useful because they can only represent linear functions, the same as 1-layer networks. However, they share many of the training phenomena with nonlinear networks, and

Linear networks can only represent linear functions, because each layer computes a linear function, and the composition of linear functions is linear.

therefore are an important tool for understanding neural net phenomena for which plain linear regression isn't an adequate model.

Assume we have a data matrix $\mathbf{X}$ which is already centered to be zero-mean. Let $\mathbf{S}^2 = \text{diag}(\sigma_1^2, \dots, \sigma_K^2)$ contain the top K eigenvalues of the empirical covariance matrix $\mathbf{\Sigma} = \frac{1}{N}\mathbf{X}^\top\mathbf{X}$ (ordered from largest to smallest), and the columns of $\mathbf{U}$ contain the corresponding eigenvectors, i.e. the principal components. For simplicity, we assume the σ_i are all distinct (this is necessary to obtain a unique solution).

Remember, the columns of $\mathbf{U}$ are orthogonal, due to the Spectral Theorem.

Recall that an autoencoder is a neural net architecture with two components: an **encoder** which maps an input $\mathbf{x}$ to a (typically low-dimensional) code vector $\mathbf{z} = f_{\text{enc}}(\mathbf{x})$, and a **decoder** which maps the latents to a reconstruction $\hat{\mathbf{x}} = f_{\text{dec}}(\mathbf{z})$. It is trained with a cost function that encourages $\hat{\mathbf{x}}$ to be close to $\mathbf{x}$, such as $\|\hat{\mathbf{x}} - \mathbf{x}\|^2$. A linear autoencoder uses a linear function for the encoder: $f_{\text{enc}}(\mathbf{x}) = \mathbf{W}_1\mathbf{x}$ and a linear function for the decoder: $f_{\text{dec}}(\mathbf{z}) = \mathbf{W}_2\mathbf{z}$. Putting this together, $\hat{\mathbf{x}} = \mathbf{W}_2\mathbf{W}_1\mathbf{x}$, and the squared error cost function is:

$$\frac{1}{2N}\sum_{i=1}^{N} \|\mathbf{W}_2\mathbf{W}_1\mathbf{x}^{(i)} - \mathbf{x}^{(i)}\|^2.$$

It's well-known that training linear autoencoders is equivalent to principal component analysis (PCA). More precisely, the optimal solutions correspond to any pair of matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ which project onto the principal subspace. One such solution is $\mathbf{W}_1 = \mathbf{U}^\top$ and $\mathbf{W}_2 = \mathbf{U}$. However, the individual principal components aren't recoverable: there exists a symmetry whereby we can map $\mathcal{T}_{\mathbf{A}}(\mathbf{W}_1, \mathbf{W}_2) = (\mathbf{A}\mathbf{W}_1, \mathbf{W}_2\mathbf{A}^{-1})$ for any invertible matrix $\mathbf{A}$, obtaining an equivalent model (in terms of the reconstruction). This transformation can be seen as transforming the latent space, i.e. if $\mathbf{z} = \mathbf{W}_1\mathbf{x}$, then $\mathcal{T}_{\mathbf{A}}(\mathbf{z}) = \mathbf{A}\mathbf{z}$. Since invertible linear transformations include things like rotations and permutations, this shows that the individual principal components aren't recoverable.

See the CSC413 lecture notes for a detailed derivation of this.

In order to break the symmetry, we'll add a somewhat unusual regularizer, **non-uniform** ℓ_2. Consider the following objective, which penalizes the squared norms of the entries of $\mathbf{W}_1$ and $\mathbf{W}_2$, except that it penalizes some rows and columns more heavily than others:

$$\frac{1}{2N}\sum_{i=1}^{N} \|\mathbf{W}_2\mathbf{W}_1\mathbf{x}^{(i)} - \mathbf{x}^{(i)}\|^2 + \tfrac{1}{2}\|\mathbf{\Lambda}^{1/2}\mathbf{W}_1\|_F^2 + \tfrac{1}{2}\|\mathbf{W}_2\mathbf{\Lambda}^{1/2}\|_F^2,$$

Here, $\|\cdot\|_F$ denotes the **Frobenius norm**, $\|\mathbf{B}\|_F = \sqrt{\text{tr}\,\mathbf{B}^\top\mathbf{B}} = \sqrt{\sum_{i,j} B_{ij}^2}$. Note also that $\|\mathbf{\Lambda}^{1/2}\mathbf{W}_1\|_F^2 = \text{tr}\,\mathbf{W}_1^\top\mathbf{\Lambda}\mathbf{W}_1$, which may be more familiar to some readers. Hence, $\|\mathbf{\Lambda}^{1/2}\mathbf{W}\|_F^2 = \sum_{i,j}\lambda_i w_{ij}^2$.

where $\mathbf{\Lambda}$ is a diagonal matrix with increasing diagonal entries $(\lambda_1, \dots, \lambda_K)$. For simplicity, we assume $\lambda_k < \sigma_k^2$ (this is necessary for all rows and columns of the optimal solution to be nonzero). It can be shown that under this assumption, the global optimum of this objective consists of the individual principal components, sorted in descending order:

$$\begin{aligned}\mathbf{W}_1^\star &= \mathbf{P}(\mathbf{I} - \mathbf{\Lambda}\mathbf{S}^{-2})^{1/2}\mathbf{U}^\top \\ \mathbf{W}_2^\star &= \mathbf{U}(\mathbf{I} - \mathbf{\Lambda}\mathbf{S}^{-2})^{1/2}\mathbf{P},\end{aligned} \tag{12}$$

where $\mathbf{P}$ is a diagonal matrix whose diagonal entries are ± 1; its presence in these formulas indicates that there remains a reflection symmetry for each of the principal components. Apart from this symmetry, the solution is

unique: the rows of $\mathbf{W}_1$ and the columns of $\mathbf{W}_2$ each contain the principal components, ordered from largest to smallest, and rescaled by a factor less than 1.

We don't need to concern ourselves with the derivation of this result (the details are a little hairy), but here's some intuition. In the absence of the regularizer, the optimal solution is to project the data onto the principal subspace, which can be achieved with $\mathbf{W}_1 = \mathbf{U}^\top$ and $\mathbf{W}_2 = \mathbf{U}$ (see the CSC413 readings on autoencoders). When we regularize the norms of the entries of $\mathbf{W}_1$ and $\mathbf{W}_2$, the optimal solution will be a compromise between reconstruction error and the norms, so we'd expect to shrink $\mathbf{W}_1$ and $\mathbf{W}_2$; this is achieved by the $(\mathbf{I} - \mathbf{\Lambda}\mathbf{S}^{-2})^{1/2}$ factor. Observe from this formula that rows or columns with heavier regularization are shrunk more than those with lighter regularization, and hence will reconstruct their respective projections less accurately. Therefore, to minimize reconstruction error, we'd like to allocate the more heavily regularized dimensions to the directions in input space with lower variance. The way to do this is to sort the principal components in decreasing order, which is exactly what Eqn. 12 does.

All of this setup is captured by the following code:

```
# Generate a random data matrix.
onp.random.seed(0)
N, D, K = 1000, 20, 10
VARIANCES = np.linspace(1, 0, D)
X = onp.random.normal(0, np.sqrt(VARIANCES), size=(N, D))
W1_init = onp.random.normal(0, 0.1, size=(K, D))
W2_init = onp.random.normal(0, 0.1, size=(D, K))

# Convenience functions for converting between matrices and flat
# parameter vectors.
w_flat_init, unflatten = ravel_pytree((W1_init, W2_init))
def flatten(tree):
    return ravel_pytree(tree)[0]

# Compute the principal components.
Sigma = X.T @ X / N
d, Q = np.linalg.eigh(Sigma)
s_sq = d[::-1][:K]
U = Q[:, ::-1][:, :K]

# Set the l2 penalties consistent with the constraint.
lambdas = np.linspace(0, .5 * s_sq[-1], K)

# Objective function.
def fobj(w_flat):
    W1, W2 = unflatten(w_flat)
    reconst = W2 @ W1 @ X.T
    return .5 * np.sum((reconst - X.T)**2) / N + \
           .5 * np.sum(lambdas.reshape((-1, 1)) * W1**2) + \
           .5 * np.sum(lambdas.reshape((1, -1)) * W2**2)
grad_fobj = grad(fobj)
```

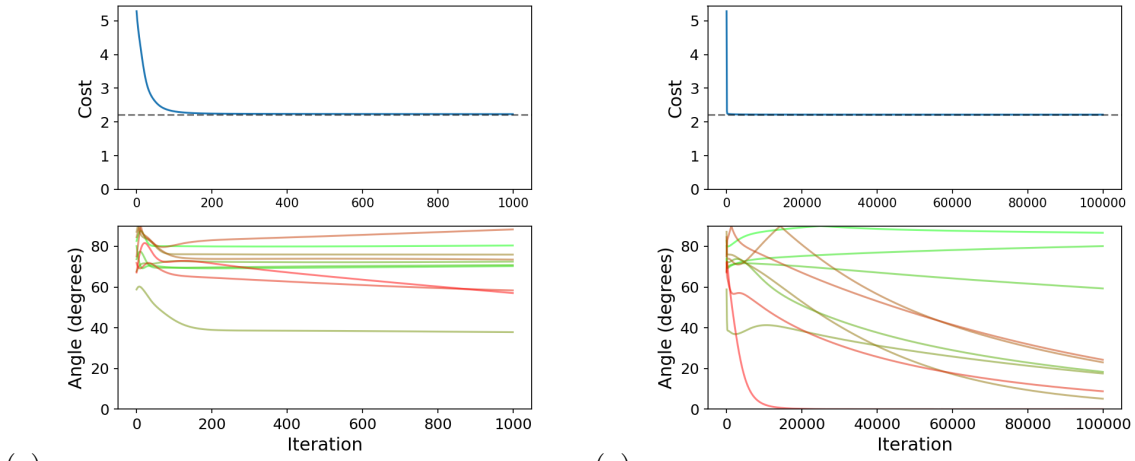


Figure 1: Convergence of the linear autoencoder in terms of the cost function **(top)** and the angle between each weight column/vector and the matching principal component **(bottom)**. The dashed line indicates the optimal cost. Principal components are color-coded from red (largest) to green (smallest). **(left)** 1000 iterations, **(right)** 100,000 iterations.

```
# Optimal weights.
W1_opt = np.sqrt(1 - lambdas/s_sq).reshape((-1, 1)) * U.T
W2_opt = np.sqrt(1 - lambdas/s_sq).reshape((1, -1)) * U
w_flat_opt = flatten((W1_opt, W2_opt))
```

Now for the interesting part: what happens when we try to optimize this objective? The results are shown in Figure 1. We can see that it very quickly achieves close to the minimum possible cost. However, it takes a very long time to learn the individual principal components. What's going on?

The problem is that the cost function is more sensitive to some directions than to others. We can understand this by measuring the curvature in various directions at the optimum. The curvature in a direction $\mathbf{v} \neq \mathbf{0}$ can be measured using the **Rayleigh quotient**:

$$\frac{\mathbf{v}^\top \mathbf{H}\mathbf{v}}{\mathbf{v}^\top \mathbf{v}},$$

which can be computed with the following code:

```
def rayleigh_quotient(J, w, v):
    Hv = hvp(J, w, v)
    return (Hv @ v) / (v @ v)
```

Recall that higher curvature directions train quickly, and low curvature directions train slowly. In general, there are a great many directions we can look at (weight space is high-dimensional), but in this case, the symmetries of the problem naturally suggest some directions.

First consider the transformation group given by rescaling $\mathcal{T}_\gamma(\mathbf{W}_1, \mathbf{W}_2) = (\gamma\mathbf{W}_1, \gamma\mathbf{W}_2)$ for a parameter $\gamma > 0$. By computing $\mathbf{v} = \mathcal{R}\mathcal{T}_\gamma(\mathbf{W}_1, \mathbf{W}_2)$, the tangent vector to this transformation group at $\gamma = 1$, we can measure the effect of slightly increasing the scale of the weights. Note that this transformation will change the reconstructions, so we'd expect it to make a significant contribution to the objective, i.e. $\mathbf{v}$ should be a high curvature direction. Observe that the tangent vector is just the Jacobian of the transformation, so we can compute it with forward mode autodiff, using the following code:

The $\mathcal{R}$ notation is introduced in Section 3.2.

You might have noticed that the tangent vector is just $(\mathbf{W}_1, \mathbf{W}_2)$, so strictly speaking this code is unnecessary. But it shows how this idea can be generalized.

```
def rescale(w_flat, gamma):
    W1, W2 = unflatten(w_flat)
    return flatten((gamma*W1, gamma*W2))

_, v_scale = jvp(lambda g: rescale(w_flat_opt, g), (1,), (1,))

print(rayleigh_quotient(fobj, w_flat_opt, v_scale))
```

This produces the output `1.3586808`.

The original objective had a rotational ambiguity, and the new objective takes a long time to learn the correct rotation. Therefore, we might conjecture that the cost function has low curvature in directions corresponding to rotations of the latent space. In particular, consider the mapping $\mathcal{T}_\theta(\mathbf{W}_1, \mathbf{W}_2) = (\mathbf{Q}_\theta\mathbf{W}_1, \mathbf{W}_2\mathbf{Q}_\theta^\top)$, where $\mathbf{Q}_\theta$ represents a **Givens rotation** of the first two rows of $\mathbf{W}_1$ (or columns of $\mathbf{W}_2$) by angle θ:

$$\mathbf{Q}_\theta = \begin{pmatrix} \cos\theta & -\sin\theta & & & \\ \sin\theta & \cos\theta & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix}$$

Similarly to the scaling transformation, we can write a few lines of JAX code that compute the tangent vector to this rotation group and then compute the curvature in that direction:

```
def block_diag(A, B):
    return np.vstack([np.hstack([A, np.zeros((A.shape[0], B.shape[1]))]),
                      np.hstack([np.zeros((B.shape[0], A.shape[1])), B])])

def rotate(w_flat, theta):
    W1, W2 = unflatten(w_flat)
    rot = np.array([[np.cos(theta), -np.sin(theta)],
                    [np.sin(theta), np.cos(theta)]])
    Q_theta = block_diag(rot, np.eye(K-2))
    return flatten((Q_theta @ W1, W2 @ Q_theta.T))

_, v_rot = jvp(lambda th: rotate(w_flat_opt, th), (0,), (1,))

print(rayleigh_quotient(fobj, w_flat_opt, v_rot))
```

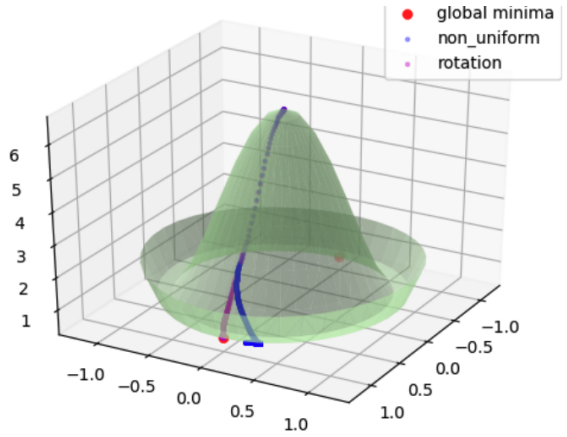


Figure 2: Figure from Bao et al. (2020) showing the loss landscape of the linear autoencoder with non-uniform ℓ_2 regularization. The radial direction corresponds to rescaling, and the angle corresponds to rotation (see main body for details). The narrow valley corresponds to ill-conditioning of the optimization landscape. (Note: this is a small example chosen to *understate* the effect for purposes of illustration.)

This produces the output `0.00041926137`.

So we can see that the curvature in the rotation direction is nearly 4 orders of magnitude smaller than the curvature in the scaling direction. This creates ill-conditioning, and the training is much slower along the rotation directions. This is an instance of **weak symmetry breaking**, whereby the constraint added to break a symmetry has only a weak effect on the optimization, and therefore it takes a long time for gradient descent to break the symmetry.

Figure 2 illustrates a subspace of the optimization landscape for $K = 2$ which includes both scaling and rotation. Specifically, it uses the transformation:

$$\begin{aligned}
\mathbf{W}_1 &= \gamma \mathbf{Q}_\theta (\mathbf{I} - \mathbf{\Lambda} \mathbf{S}^{-2})^{1/2} \mathbf{U}^\top \\
\mathbf{W}_2 &= \mathbf{W}_1^\top . \\
\mathbf{Q}_\theta &= \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} .
\end{aligned}$$

The xy-coordinate is given by $(\gamma \cos\theta, \alpha \sin\theta)$. Note that this space includes the global optimum (Eqn. 12). Note also that the transformation is isometric, i.e. it faithfully represents distances. There is a high curvature in the radial direction, corresponding to rescaling, while there is very small curvature in the rotation direction. In general, ill-conditioned curvature manifests as narrow valleys like this one, although typically it is hard to visualize due to the high dimensionality of the optimization problem.

To sum up: gradient descent takes a *really* long time to learn the correct rotation of the principal components, and this is reflected in the curvature around the optimum (as measured by Rayleigh quotients of the Hessian). How typical is this of neural net optimization more generally? On one

hand, it's a toy example without direct practical value — computing the principal components of small datasets is a trivial operation. There are other regularizers and iterative algorithms which make the linear autoencoder converge much faster (see Bao et al. (2020)).

On the other hand, it's one of the few model systems we have to understand *representation learning*, i.e. the situation where we don't just care about making good predictions, but also desire some property of the latent representation. When we train an object recognizer on ImageNet, often we don't just want to minimize the classification loss, but we also want to learn a generally useful representation that can be transferred to other tasks. The latter problem is much harder to quantify than the former. It's been observed that classification conv nets need to be trained with much higher learning rates at the start of training than are needed to achieve good training accuracy, a phenomenon which is still poorly understood. I wouldn't be terribly surprised if the eventual explanation has something to do with low curvature in the directions relevant to learning good representations, just like in this toy example.

4.4 The Gauss-Newton Hessian

We rarely work with the true Hessian of neural networks, even through implicit matrix-vector products. One reason is that many of our architectures are not twice differentiable, e.g. because we use the ReLU activation function. If we ask JAX for the second derivative of ReLU, it will return the answer 0 without complaining, but this obviously isn't "correct", and might not be a good local approximation to the cost. Another problem is that $\mathbf{H}$ might have negative eigenvalues away from the optimum, and there are a lot of use cases that require a positive semidefinite matrix (such as approximate inversion using conjugate gradients; see below).

Recall that the cost for one training example is the composition of the network function with a loss function (defined on the outputs, typically simple and convex): $\mathcal{J}_{\mathbf{x},t}(\mathbf{w}) = \mathcal{L}(f(\mathbf{w}, \mathbf{x}), \mathbf{t})$. Note that there are often multiple ways we can define the network function and loss function. For instance, in classification, we could choose $\mathbf{z} = f(\mathbf{w}, \mathbf{x})$ to be the logits and $\mathcal{L}(\mathbf{z}, \mathbf{t})$ to be softmax-cross-entropy loss, or we could define $\mathbf{y} = f(\mathbf{w}, \mathbf{x})$ to be the probabilities and $\mathcal{L}(\mathbf{y}, \mathbf{t})$ to be cross-entropy. This distinction turns out to be significant, and throughout this course we will generally use $\mathbf{z} = f(\mathbf{w}, \mathbf{x})$ as the logits and $\mathcal{L}(\mathbf{z}, \mathbf{t})$ as softmax-cross-entropy loss.

The Hessian can be decomposed as follows:

$$\nabla^2 \mathcal{J}_{\mathbf{x},t}(\mathbf{w}) = \mathbf{J}_{\mathbf{zw}}^\top \mathbf{H}_\mathbf{z} \mathbf{J}_{\mathbf{zw}} \; + \; \sum_a \frac{\partial \mathcal{L}}{\partial y_a} \nabla^2_\mathbf{w} [f(\mathbf{x}, \mathbf{w})]_a. \tag{13}$$

where $\mathbf{H}_\mathbf{z} = \nabla^2_\mathbf{z} \mathcal{L}(\mathbf{z}, \mathbf{t})$ is the **output Hessian**. The first term involves first derivatives of the network and second derivatives of the loss function; the second term involves second derivatives of the network and first derivatives of the cost function. If we simply drop the second term, we get the **Gauss-Newton Hessian**, typically denoted as $\mathbf{G}$:

$$\mathbf{G} = \mathbf{J}_{\mathbf{zw}}^\top \mathbf{H}_\mathbf{z} \mathbf{J}_{\mathbf{zw}}. \tag{14}$$

Unfortunately, terminology in ML papers is inconsistent. Sometimes $\mathbf{G}$ is just called the Hessian. And sometimes the term *Gauss-Newton matrix* is used to refer to $\mathbb{E}[\mathbf{J}_{\mathbf{zw}}^\top \mathbf{J}_{\mathbf{zw}}]$, i.e. the output space Hessian is dropped. This terminology comes from the classical Gauss-Newton optimization algorithm, which assumed squared error loss (hence the output Hessian was the identity matrix).

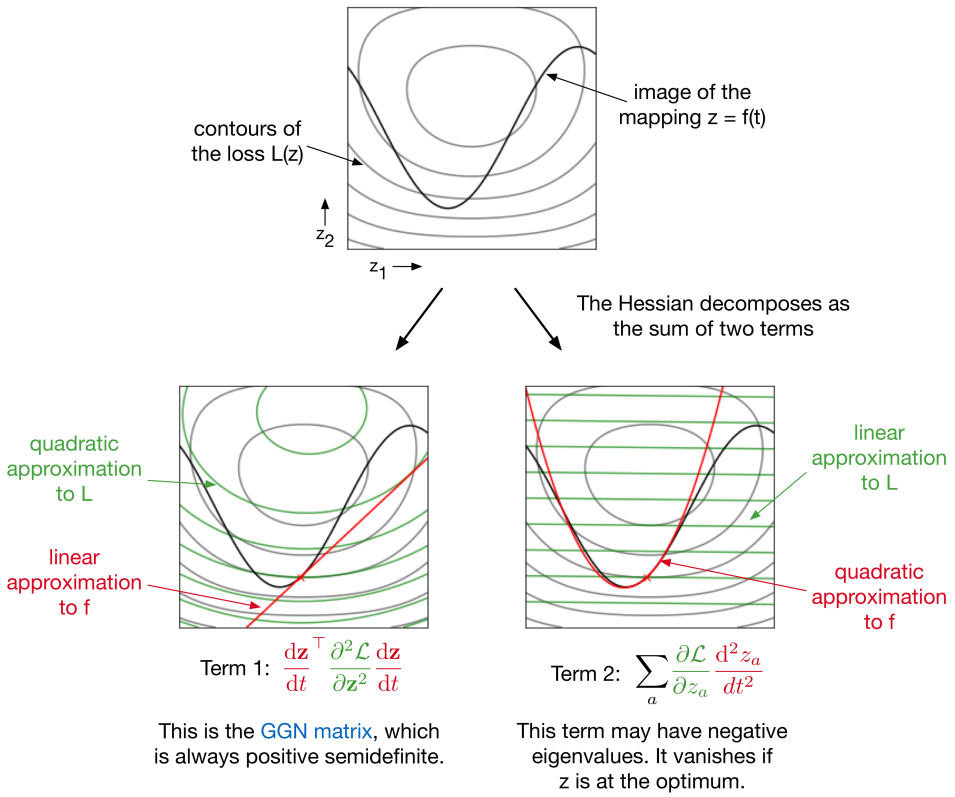


Figure 3: Illustration of the Gauss-Newton approximation to the Hessian.

One way to understand the Gauss-Newton Hessian is that we **linearize** the network around the current weights, i.e. apply the first-order Taylor approximation:

$$f_{\text{lin}}(\mathbf{w}', \mathbf{x}) = f(\mathbf{w}, \mathbf{x}) + \mathbf{J}_{\mathbf{zw}}(\mathbf{w}' - \mathbf{w}).$$

JAX provides a function called `linearize` which constructs this linearized approximation, although it's also straightforward to do directly from JVPs.

Then $\mathbf{G}$ is the Hessian of the cost function for the linearized model, i.e. $\mathcal{J}_{\text{lin},\mathbf{x},\mathbf{t}}(\mathbf{w}') = \sum_{i=1}^N \mathcal{L}(f_{\text{lin}}(\mathbf{w}', \mathbf{x}), \mathbf{t})$. This is illustrated in Figure 3. Linearizing the network sounds like a pretty naïve thing to do, but we'll see later in the course that it's a surprisingly powerful trick for understanding certain phenomena.

Another justification for this linearization is that because the second term includes the first derivatives of the loss, it vanishes whenever the loss is minimized individually for each training example. E.g., in the case of linear regression, this happens if every training example is fit perfectly. Note that this requirement is stronger than simply assuming $\mathbf{w}$ is at the optimum; we're requiring that the loss on *each individual* training example be minimized with respect to its outputs. Hence, for networks whose capacity is too small to fit the training set, the Gauss-Newton approximation will be inexact, even at the global optimum.

Unlike $\mathbf{H}$, $\mathbf{G}$ requires only first derivatives of the network function, so we're free to apply it to ReLU networks. Also, as long as the loss function is convex (as a function of the outputs), $\mathbf{G}$ is guaranteed to be positive semidefinite.

Technically ReLU isn't even once differentiable. But in machine learning, we're generally perfectly happy to compute first derivatives of ReLU. Second derivatives of ReLU are nonsensical, though.

Just like with $\mathbf{H}$, it's impractical to represent $\mathbf{G}$ explicitly, so we typically work with matrix-vector products. Multiplying by $\mathbf{G}$ involves 3 steps: multiplying by $\mathbf{J}_{\mathbf{zw}}$, multiplying by $\mathbf{H}_{\mathbf{z}}$, and multiplying by $\mathbf{J}_{\mathbf{zw}}^\top$. This can be achieved in a few lines of code:

Exercise: prove that if a symmetric matrix $\mathbf{A}$ is positive semidefinite, then $\mathbf{BAB}^\top$ is symmetric and positive semidefinite, for any matrix $\mathbf{B}$ (of appropriate dimensions).

```
def gnhvp(f, L, w, v):
    z, R_z = jvp(f, (w,), (v,))
    R_gz = hvp(L, z, R_z)
    _, f_vjp = vjp(f, w)
    return f_vjp(R_gz)[0]
```

Just as with all of our other implicit matrix-vector products, the time cost of this is linear in the cost of a forward pass.

If you check carefully, this code isn't as efficient as it could be, because it computes the forward pass through the network twice. Can you find a way to eliminate this redundant forward pass?

As it turns out, there's a second option for efficiently computing with $\mathbf{G}$, one which wasn't available to us for $\mathbf{H}$: the Pullback Sampling Trick. We'll get to this next week when we talk about pullback metrics, since the trick is more natural to describe in that context.

5 Approximately Solving Linear Systems

We've been working with matrices that are too large to represent explicitly, and so far we've been working with them entirely through implicit matrix-vector products (MVPs). This might seem a bit limiting, but there's actually an astonishing amount you can do with just matrix-vector products. Having a large matrix you can only access through MVPs is a frequent occurrence in scientific computing, and researchers have come up with ingenious algorithms based on MVPs. One such algorithm is **conjugate gradient (CG)**, an algorithm for approximately solving a linear system involving a positive definite matrix.

More specifically, suppose we have a positive definite matrix $\mathbf{A}$, and we'd like to approximately solve $\mathbf{Ax} = \mathbf{b}$. This can be formulated as the optimum of the following optimization problem:

Note that this objective only has a unique minimum if it is strictly convex, hence the requirement that $\mathbf{A}$ be positive definite.

$$\mathcal{J}(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top \mathbf{Ax} - \mathbf{b}^\top \mathbf{x}. \tag{15}$$

The CG algorithm itself is rather hairy, but fortunately we don't need to get into the weeds in order to reason about it. The main concept to understand is the **Krylov subspace** $\mathcal{K}_k(\mathbf{A}, \mathbf{r})$ for a vector $\mathbf{r}$, defined as:

$$\mathcal{K}_k(\mathbf{A}, \mathbf{r}) = \text{span}\{\mathbf{r}, \mathbf{Ar}, \ldots, \mathbf{A}^{k-1}\mathbf{r}\}. \tag{16}$$

To make sense of this subspace, observe that if $\mathbf{x} \in \mathcal{K}_k(\mathbf{A}, \mathbf{r})$, then $\mathbf{Ax} \in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r})$. Hence, $\mathcal{K}_k(\mathbf{A}, \mathbf{r})$ is the set of vectors obtainable, starting from $\mathbf{r}$, through linear combinations and at most $k-1$ multiplications by $\mathbf{A}$.

Similarly, observe that if $\mathbf{x} \in \mathcal{K}_k(\mathbf{A}, \mathbf{b})$, then $\nabla\mathcal{J}(\mathbf{x}) \in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{b})$. So consider an iterative optimization algorithm where $\mathbf{x}^{(0)} = \mathbf{0}$ and each iterate $\mathbf{x}^{(k)}$ is somehow computed as a linear combination of all the past iterates and gradients (i.e. $\mathbf{x}^{(k)} = \sum_{\ell=1}^{k} \alpha_\ell \mathbf{x}^{(k-\ell)} + \beta_\ell \nabla\mathcal{J}(\mathbf{x}^{(k-\ell)})$). Note that this formulation includes gradient descent, as well as various commonly used extensions (which we'll cover later in the course) such as heavy ball momentum, Nesterov Accelerated Gradient, and iterate averaging. A simple inductive argument shows that $\mathbf{x}^{(k)} \in \mathcal{K}_k(\mathbf{A}, \mathbf{b})$ for all k.

CG is an algorithm for minimizing Eqn. 15 with the rather magical property that the kth iteration exactly minimizes $\mathcal{J}$ within the Krylov subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$:

$$\mathbf{x}^{(k)} = \underset{x \in \mathcal{K}_k(\mathbf{A},\mathbf{b})}{\arg\min}\ \mathcal{J}(\mathbf{x}). \tag{17}$$

Moreover, it does this using a *single* MVP by $\mathbf{A}$ per iteration (as well as linear combinations, which are cheap), and stores only a *small constant* number of vectors in memory which are the same dimension as $\mathbf{x}$. Since the kth iterate is computed using k MVPs, and any iterate computed using k MVPs must lie within $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$, this implies that CG minimizes the cost *as fast as is possible for a given number of MVPs!* It's pretty surprising that this minimization can be done with low computational overhead, and also without storing the previous iterates and/or gradients in memory. The reasons this all works are beyond the scope of this class, but see Shewchuk (1994) for a readable tutorial.

That is to say, CG can't be beat without exploiting some other source of information not obtainable from MVPs or gradients.

The best way to run CG is to call a library routine. One gotcha is to make sure the routine is *linear CG*, as opposed to *nonlinear CG*, a set of CG-like algorithms used to minimize non-quadratic functions. Hence, if using SciPy, the right routine is `scipy.sparse.linalg.cg`, rather than `scipy.optimize.minimize(method='CG')`. The following code is a convenient wrapper around the SciPy routine:

```
def approx_solve(A_mvp, b, niter):
    dim = b.size
    A_linop = scipy.sparse.linalg.LinearOperator((dim,dim), matvec=A_mvp)
    res = scipy.sparse.linalg.cg(A_linop, b, maxiter=niter)
    return res[0]
```

Another gotcha is that CG can be numerically sensitive, so it's a good idea to do the computations in `float64`, at least for debugging purposes. (By default, JAX and other deep learning frameworks use `float32` for computational and memory efficiency.) To tell JAX to enable `float64`, we need to add the following to our preamble (before calling any JAX functions):

```
from jax.config import config
config.update('jax_enable_x64', True)
```

Once this is done, you can pass in `float64` as the `dtype` when creating an array, just like in NumPy. The following convenience function also converts a whole data structure to `float64`:

```
def make_float64(params):
    return tree_map(lambda x: x.astype(np.float64), params)
```

5.1 Example: Sensitivity to Dataset Perturbations

A neat application of the techniques we've just developed is to understanding how the optimal solution to an optimization problem changes if we slightly perturb various aspects of the problem such as the data or hyperparameters. E.g., if a neural network is fooled by an adversarial input, we might want to figure out which of the inputs are responsible for that prediction. One way to do this is to measure the effect of small changes to a

dataset, such as perturbing a label or an input location, or changing the weighting of the training examples. The influence function measures how a model's predictions change as a function of some aspect of the training data, we can use the techniques of this lecture to compute linearized approximations to it, in order to avoid re-training our model many times from scratch. The most well-known instance of this in the field of deep learning concerns **influence functions**, which estimate how a network's predictions will change if a training example (or group of training examples) is removed. Influence functions were developed in statistics decades ago and introduced to our field by Koh and Liang (2017), who used them to explain a model's predictions, diagnose adversarial examples, detect mislabeled training examples, and carry out data poisoning attacks.

Assume we've found the minimum $\mathbf{w}_\star = \arg\min_{\mathbf{w}} \mathcal{J}(\mathbf{w}; \boldsymbol{\theta})$ to an unconstrained optimization problem whose cost function is parameterized by a vector $\boldsymbol{\theta}$ (for instance, weights on the training examples or a set of hyperparameters). We're interested in how the optimal solution depends on $\boldsymbol{\theta}$, so we define $\mathbf{w}_\star = r(\boldsymbol{\theta})$ to emphasize the functional dependency. The function $r(\boldsymbol{\theta})$ is called the **response function**, or **rational reaction function**, and the **Implicit Function Theorem (IFT)** guarantees its existence under certain conditions that we won't worry about for now. For simplicity, assume $\mathcal{J}$ is twice differentiable and $\mathbf{H}$ is strictly positive definite at $\mathbf{w}_\star$.

The constrained case can be handled using similar ideas.

We are interested in computing the Jacobian $\mathbf{J}_{\mathbf{w}_\star \boldsymbol{\theta}}$ of the response function $r(\boldsymbol{\theta})$, which is known as the **response Jacobian**, or **reaction Jacobian**. We'll become very good friends with the response Jacobian at the end of the course, when we cover bilevel optimization. Once we get there, we'll see how to derive $\mathbf{J}_{\mathbf{w}_\star \boldsymbol{\theta}}$, but for now please take my word for it that the formula is:

$$\mathbf{J}_{\mathbf{w}_\star \boldsymbol{\theta}} = -\left[\nabla^2_{\mathbf{w}} \mathcal{J}(\mathbf{w}; \boldsymbol{\theta})\right]^{-1} \nabla^2_{\mathbf{w}\boldsymbol{\theta}} \mathcal{J}(\mathbf{w}; \boldsymbol{\theta}). \tag{18}$$

Hopefully the 1-D example illustrated in Figure 4 will convince you that this formula is at least plausible. The figure shows $\mathcal{J}(w; \lambda) = g(w) + \lambda w$, evaluated at $\lambda = 0$ (blue) and $\lambda = 3$ (orange). Here, $\nabla^2_{w\lambda} \mathcal{J}(w; \lambda) = \partial^2 \mathcal{J} / \partial w \partial \lambda = 1$. The curvature $\nabla^2_w \mathcal{J}(w; \lambda) = \partial^2 \mathcal{J} / \partial w^2$ is positive at both local minima. Increasing λ shifts the local minima to the left, indicating that Eqn. 18 has the correct sign. The flatter minimum is shifted more than the sharper one, consistent with the dependence on the inverse of the curvature.

Now let's apply everything we've learned so far in this lecture to implementing matrix-vector products with $\mathbf{J}_{\mathbf{w}_\star \boldsymbol{\theta}}$ (Eqn. 18). We'll consider the dependency of the optimal weights on the training labels (although the same techniques apply to other kinds of dataset perturbations as well). Hence, the hyperparameter $\boldsymbol{\theta}$ is simply $\mathbf{t}$, the vector of training labels. So here is our parameterized cost:

```
def parameterized_cost(w, t):
    return L(f_net(w, x), t)
```

The mixed second derivative $\nabla^2_{\mathbf{wt}} \mathcal{J}(\mathbf{w}; \mathbf{t})$ is something we haven't seen before, but computationally it's pretty innocuous: we can compute MVPs analogously to the Hessian. We use the forward-over-reverse strategy (see

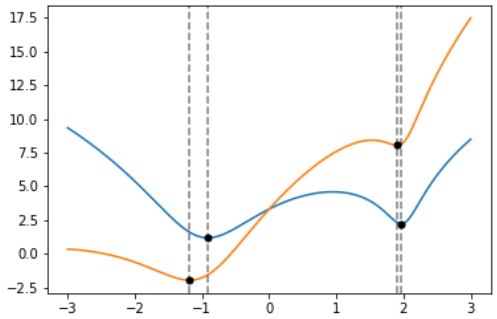


Figure 4: Illustration of the formula for the response Jacobian (Eqn. 18). Shows $\mathcal{J}(w;\lambda) = g(w) + \lambda w$ for $\lambda = 0$ (blue) and $\lambda = 3$ (orange). See main body for description.

Section 4.2), computing the directional derivative of $\nabla_{\mathbf{w}}\mathcal{J}(\mathbf{w};\mathbf{t})$ with respect to $\mathbf{t}$:

```
def mixed_second_mvp(w, t, R_t):
    grad_cost = grad(parameterized_cost, 0)       # gradient w.r.t. w
    grad_cost_t = lambda t: grad_cost(w, t)
    return jvp(grad_cost_t, (t,), (R_t,))[1]      # forward-over-reverse
```

The term $\left[\nabla^2_{\mathbf{w}}\mathcal{J}(\mathbf{w};\boldsymbol{\theta})\right]^{-1}$ is the inverse of $\mathbf{H}$, evaluated at $\mathbf{w}_\star(\boldsymbol{\theta})$. Because of the inverse, we need to be able to solve a linear system involving $\mathbf{H}$. Note that, in order to apply CG, the matrix must be positive definite. The Hessian $\mathbf{H}$ isn't positive definite in general, but we are at least guaranteed that it is positive semidefinite at a (local) minimum, and we can add a small multiple of $\mathbf{I}$ to make it positive definite. If we haven't actually reached a (local) minimum, we might prefer to substitute the GN Hessian $\mathbf{G}$ to ensure the linear system we're solving is positive definite. So here is how we approximately solve the linear system:

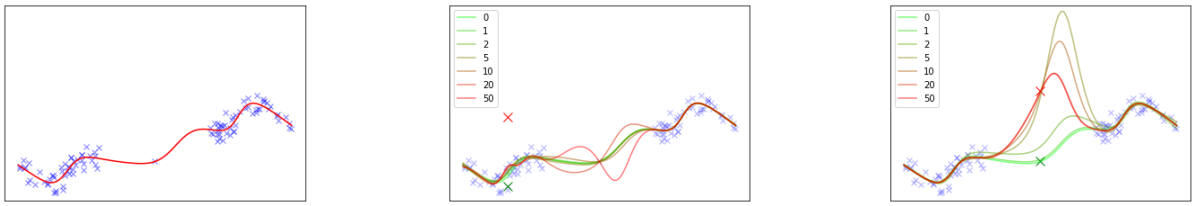
Koh and Liang (2017) actually solved the linear system using another method called stochastic Neumann iterations, rather than conjugate gradient. We'll see why this is a good idea in Chapter 7.

```
def dampen(mvp, lam):
    def new_mvp(x):
        return mvp(x) + lam*x
    return new_mvp

def approx_solve_H(w, Rg_w, lam, niter):
    mvp = lambda v: gnhvp(lambda w: f_net(w, x), lambda y: L(y, t), w, v)
    mvp_damp = dampen(mvp, lam)
    return approx_solve(mvp_damp, Rg_w, niter)
```

According to Eqn. 18, to compute an MVP with $\mathbf{J}_{\mathbf{w}_\star\boldsymbol{\theta}}$, we just do these two operations in sequence:

```
def response_jacobian_vector_product(w, t, R_t, lam, niter):
    Rg_w = mixed_second_mvp(w, t, R_t)
    return approx_solve_H(w, -Rg_w, lam, niter)
```



(a) (b) (c)

Figure 5: Illustration of influence functions for a 1-D regression dataset. **(a)** the initial fit to the data. **(b, c)** Linearized effect of perturbations to two data points, moving the green $\times$ to the red $\times$. The linearized predictions are shown for varying numbers of CG iterations, from 0 (equivalent to the original fit) to 50.

This MVP tells us how the optimal weights will change in response to a small perturbation to $\mathbf{t}$. To understand what effect this has on the network's predictions, we simply compute the directional derivative of the network's predictions in this direction, using the now-familiar JVP:

```
R_w = response_jacobian_vector_product(w_opt, t, R_t, LAM, niter)
R_y = jvp(lambda w: f_net(w, x_in), (w_opt,), (R_w,))[1]
```

All of this is shown in Figure 5. Here, we fit a small tanh MLP to a 1-D regression dataset, where the data points belong to two clusters, plus another outlier point in the middle. We use our code to compute the sensitivity of the predictions to the labels of two of the data points. I.e., we compute the linearized response to the perturbation shown by moving the green $\times$ to the red $\times$. When we perturb a training example within one of the clusters, it has only a slight effect on the predictions within the clusters, since it's competing with a lot of other training examples. It does affect the predictions quite a bit in between the clusters, indicating that the predictions in those regions are unreliable. When we perturb the outlier training example, we see that it has an outsized influence on the predictions in the intermediate region.

6 Tidying Up the Alphabet Soup

By taking Taylor approximations, we've arrived at several matrices which are useful for analyzing neural nets. We're going to see quite a few more in the upcoming lectures, so all the matrices are conveniently summarized in Figure 6. The ones we haven't gotten to yet are grayed out.

References

Ryan P. Adams, Jeffrey Pennington, Matthew J. Johnson, Jamie Smith, Yaniv Ovadia, Brian Patton, and James Saunderson. Estimating the spectral density of large implicit matrices. arXiv:1802.03451, 2018.

Xuchan Bao, James Lucas, Sushant Sachdeva, and Roger Grosse. Regularized linear autoencoders recover the principal components, eventually. arXiv:2007.06731, 2020.

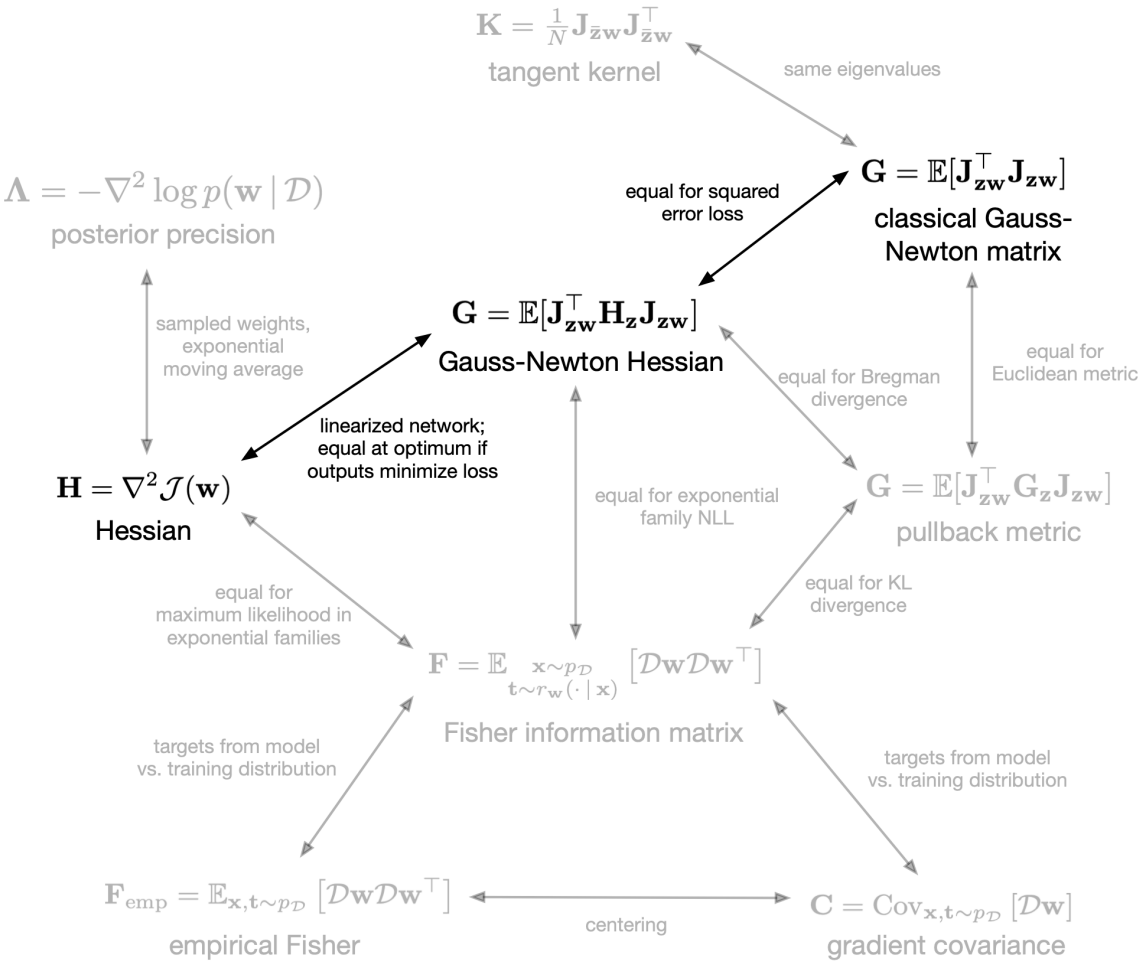


Figure 6: A summary of the relationships between the matrices used in this course. Items yet to be covered are grayed out.

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

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, 2017.

Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.