CSC 2541: Neural Net Training Dynamics

Lecture 2 - Taylor Approximations

Roger Grosse

University of Toronto, Winter 2022

Jacobian Matrix

- Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be differentiable at $\mathbf{x}_0$, and $\mathbf{y} = f(\mathbf{x})$.
- Taylor's Theorem implies that f can be approximated by its first-order Taylor approximation, or linearization:

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \mathbf{J}_{\mathbf{yx}}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + o(\|\mathbf{x} - \mathbf{x}_0\|),$$

or

$$\Delta\mathbf{y} = \mathbf{J}_{\mathbf{yx}}\Delta\mathbf{x} + o(\|\Delta\mathbf{x}\|).$$

- $\mathbf{J}_{\mathbf{yx}}(\mathbf{x}_0)$ is the Jacobian matrix of f at $\mathbf{x}_0$:

$$[\mathbf{J}_{\mathbf{yx}}(\mathbf{x}_0)]_{ij} = \frac{\partial y_i}{\partial x_j}\bigg|_{\mathbf{x}_0}$$

Typically we drop the explicit argument and just write $\mathbf{J}_{\mathbf{yx}}$, assuming it's clear from context.

Vector Form

Examples

- Matrix-vector product

$$\mathbf{z} = \mathbf{W}\mathbf{x} \qquad \mathbf{J}_{\mathbf{zx}} = \mathbf{W}$$

- Elementwise operations

$$\mathbf{y} = \exp(\mathbf{z}) \qquad \mathbf{J}_{\mathbf{yz}} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix}$$

- Note: we rarely explicitly construct the Jacobian. It's usually simpler and more efficient to directly compute matrix-vector products.

$$\mathbf{J}_{\mathbf{yz}}\mathbf{v} = \exp(\mathbf{z}) \circ \mathbf{v}$$

Jacobian Matrix

- The gradient is an important special case.
- If $f : \mathbb{R}^m \to \mathbb{R}$, then $\mathbf{J}_{y\mathbf{x}} = (\nabla y(\mathbf{x}))^\top$. (By convention, we treat $\nabla y(\mathbf{x})$ as a column vector.)
- First-order Taylor approximation to a cost function $\mathcal{J}(\mathbf{w})$:

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

- Computed using backpropagation, or reverse mode autodiff. Provided as `jax.grad`.

Jacobian Matrix

- The directional derivative, or Gateaux derivative, or R-operator, approximates the effect of a small perturbation to the input:

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

  where

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

- Computed using forward mode autodiff. Provided as `jax.jvp`. (JVP = Jacobian-vector product)

Jacobian Matrix

- The Jacobian matrix can be very large. E.g., $\mathbf{J}_{\mathbf{yw}}$ for a neural net
- So avoid representing it explicitly (except in the case of the gradient). Instead, express your algorithm in terms of Jacobian-vector products (JVPs) and vector-Jacobian products (VJPs).
  - JVPs compute $\mathbf{J}\mathbf{v}$ for a vector $\mathbf{v}$. These are basically directional derivatives (see previous slide).
  - VJPs compute $\mathbf{J}^\top\mathbf{v}$. This is the building block of backprop (see CSC2516 lectures on backprop & autodiff)
- JVPs and VJPs can both be computed in linear time using a backprop-like algorithm.
  - **Rule-of-thumb:** a JVP or VJP is between 1 and 2 times as expensive as computing $f(\mathbf{x})$.
  - VJPs (i.e. backprop) requires storing intermediate activations in memory. JVPs don't require much memory.

Jacobian Matrix

- `jax.grad` is implemented behind the scenes as a VJP.
- $\nabla \mathcal{J}(\mathbf{w}) = \mathbf{J}^\top$, so we compute a VJP with the length-1 vector (1).
- Simplified implementation:

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

Jacobian Matrix

- Perhaps the most elegant 3 lines of code I've ever seen: implementing JVP using VJP.
- **Observation:** the Jacobian of the VJP function, $g(\mathbf{v}) = \mathbf{J}^\top \mathbf{v}$, is just $\mathbf{J}^\top$.
- So we can compute $\mathbf{J}\mathbf{v}$ by calling a VJP on the VJP!

```
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]
```

- **The catch:** this implementation is only efficient in a framework (like JAX) that aggressively optimizes the computations.

(You know enough to do Problem 2.)

Second-Order Taylor Approximations

Hessian Matrix

- The Hessian matrix of a twice-differentiable function $\mathcal{J}$ at a point $\mathbf{w}_0$ is the matrix of second derivatives:

$$\mathbf{H}(\mathbf{w}_0) = \nabla^2 \mathcal{J}(\mathbf{w}_0)$$
$$H_{ij} = \frac{\partial^2 \mathcal{J}}{\partial w_i w_j}\bigg|_{\mathbf{w}=\mathbf{w}_0}$$

- $\mathbf{H}$ is symmetric because

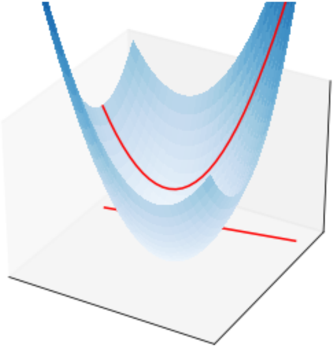
$$\frac{\partial^2 \mathcal{J}}{\partial w_i \partial w_j} = \frac{\partial^2 \mathcal{J}}{\partial w_j \partial w_i}.$$

- Second-order Taylor approximation to $\mathcal{J}$:

$$\begin{aligned}\mathcal{J}(\mathbf{w}) = \mathcal{J}(\mathbf{w}_0) + \nabla \mathcal{J}(\mathbf{w}_0)^\top (\mathbf{w} - \mathbf{w}_0) + \\ + \tfrac{1}{2}(\mathbf{w} - \mathbf{w}_0)^\top \mathbf{H} (\mathbf{w} - \mathbf{w}_0) + o(\|\mathbf{w} - \mathbf{w}_0\|^2)\end{aligned}$$

Hessian Matrix

- The Hessian measures the curvature of the function.
- The Rayleigh quotient $\mathbf{v}^\top \mathbf{H}\mathbf{v}/\|\mathbf{v}\|^2$ measures how fast the function curves up or down if you move in the direction $\mathbf{v}$.

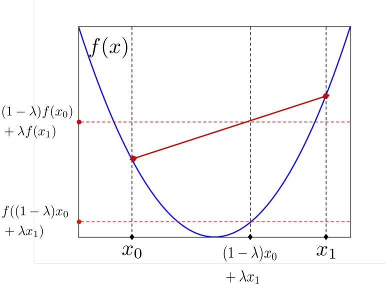


Hessian Matrix

- **Recall:** A symmetric matrix $\mathbf{A}$ is positive definite, written $\mathbf{A} \succ \mathbf{0}$, if $\mathbf{v}^\top \mathbf{A} \mathbf{v} > 0$ for any vector $\mathbf{v} \neq \mathbf{0}$.
  - Equivalently, all of $\mathbf{A}$'s eigenvalues are positive.
  - If the inequality isn't necessarily strict, then $\mathbf{A}$ is positive semidefinite (PSD), written $\mathbf{A} \succeq \mathbf{0}$.
- **Recall:** A function f is convex if:

$$\mathcal{J}(\lambda \mathbf{w}_1 + (1-\lambda)\mathbf{w}_0) \leq \lambda \mathcal{J}(\mathbf{w}_1) + (1-\lambda)\mathcal{J}(\mathbf{w}_0) \qquad \forall \mathbf{w}_0, \mathbf{w}_1, \lambda \in [0,1].$$

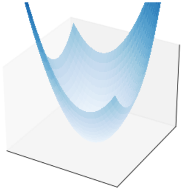
- A twice differentiable function f is convex iff its Hessian $\mathbf{H}$ is PSD.
- If $\mathbf{H} \succeq \mu \mathbf{I}$ for some μ, then it is strongly convex with parameter μ.

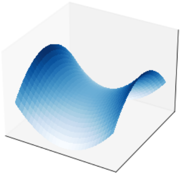


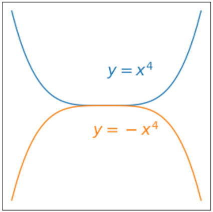
Hessian Matrix

Categorizing stationary points using the spectrum of $\mathbf{H}$

- $\mathbf{H}$ positive definite: local minimum
- $\mathbf{H}$ negative definite: local maximum (this is unusual)
- $\mathbf{H}$ has positive and negative eigenvalues: saddle point (this is more common)
- $\mathbf{H}$ is PSD but some eigenvalues are 0: could be a maximum or minimum (or neither)







Gradient Descent Dynamics

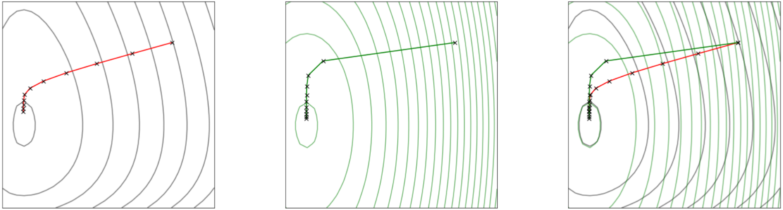
- Close to a stationary point $\mathbf{w}_\star$ we can understand the gradient dynamics using the second-order Taylor approximation

$$\mathcal{J}(\mathbf{w}) \approx \mathcal{J}(\mathbf{w}_\star) + \tfrac{1}{2}(\mathbf{w} - \mathbf{w}_\star)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_\star)$$

- This reduces it to the quadratic case from Lecture 1. Gradient descent equations (full and rotated/coordinatewise):

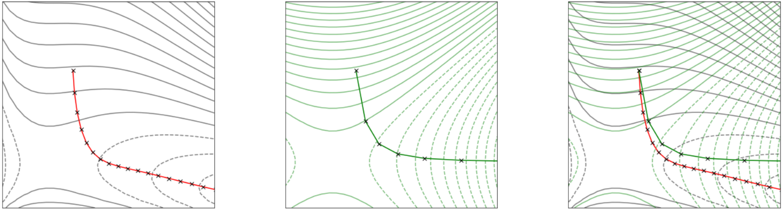
$$\begin{aligned}\mathbf{w}^{(k)} &= \mathbf{w}_\star + (\mathbf{I} - \alpha\mathbf{H})^k(\mathbf{w}^{(0)} - \mathbf{w}_\star)\\ \tilde{w}_i^{(k)} &= \tilde{w}_{i\star} + (1 - \alpha\tilde{h}_i)^k(\tilde{w}_i^{(0)} - \tilde{w}_{i\star}),\end{aligned}$$

Gradient Descent Dynamics: Local Minimum



- Stable if $\alpha < 2\tilde{h}_{\max}^{-1}$
- Speed of convergence along an eigendirection is proportional to $\tilde{h}_j$
    - **Note:** Slower convergence in a low curvature direction isn't necessarily *bad*. This depends if it contains much signal.

Gradient Descent Dynamics



- Gradient descent moves away from saddle points (and then the second order approximation is no longer accurate)
- Saddle points generally aren't a bottleneck in practice for neural net training, with the exception of symmetric initializations
- Other optimizers (e.g. Newton's method) can get stuck in saddles.

Computing with the Hessian

- The Hessian is huge, so we want to avoid constructing it explicitly.
- Instead, we write our algorithms in terms of Hessian-vector products (HVPs). I.e., compute $\mathbf{Hv}$ for a vector $\mathbf{v}$.
- **Key insight:** defining $g(\mathbf{w}) = \nabla \mathcal{J}(\mathbf{w})$, then $\mathbf{H}$ is just the Jacobian of g.
- This leads to an HVP implementation called forward-over-reverse:

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

Estimating Hessian Eigenspectra

- What do Hessian spectra of neural nets look like in practice? This is surprisingly hard to answer.
- Ghorbani et al. (2019) estimate eigenspectra using stochastic Lanczos quadrature, an HVP-based algorithm similar to conjugate gradient (covered later today)

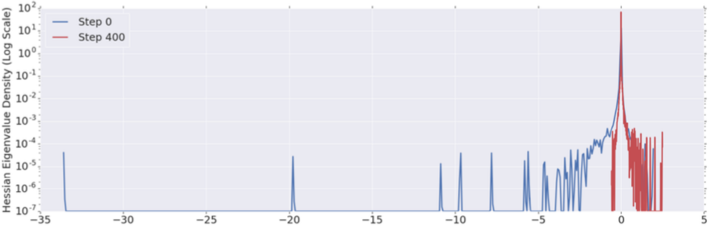


Figure 2: The evolution of the spectrum of a Resnet-32 in the beginning of training. After just 400 momentum steps, large negative eigenvalues disappear.

- **The catch:** we don't have fine-grained information about eigenvalues close to 0, and it's important to know how many eigenvalues are small vs. extremely small.

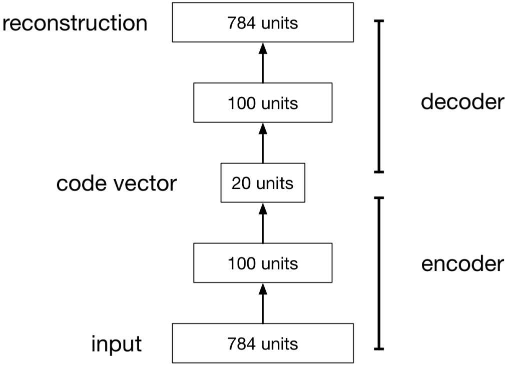
Example: Weak Symmetry Breaking in Regularized Linear Autoencoders

Example: Regularized Linear Autoencoders

- Using the Hessian to understand GD dynamics is only mathematically justified near a (local) optimum, but it can provide insight even when the Taylor approximation isn't accurate.
- Linear networks are multilayer networks with the identity activation function.
  - They can only represent linear functions, so we can often determine the optima analytically.
  - But the GD dynamics are nonlinear, and share much in common with nonlinear networks.
  - **Note:** these networks are linear as a function of the inputs, *not* as a function of the weights!

Recap: Autoencoders

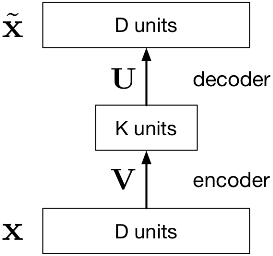
- An autoencoder is a feed-forward neural net whose job it is to take an input $\mathbf{x}$ and predict $\mathbf{x}$.
- To make this non-trivial, we need to add a bottleneck layer whose dimension is much smaller than the input.



Recap: Linear Autoencoders and PCA

- The simplest kind of autoencoder has one hidden layer, linear activations, and squared error loss.

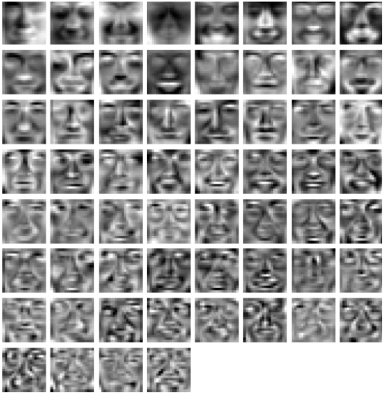
$$\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}}) = \|\mathbf{x} - \tilde{\mathbf{x}}\|^2$$



- This network computes $\tilde{\mathbf{x}} = \mathbf{U}\mathbf{V}\mathbf{x}$, which is a linear function.
- If $K \geq D$, we can choose $\mathbf{U}$ and $\mathbf{V}$ such that $\mathbf{U}\mathbf{V}$ is the identity. This isn't very interesting.
- But suppose $K < D$:
  - $\mathbf{V}$ maps $\mathbf{x}$ to a K-dimensional space, so it's doing dimensionality reduction.
  - The output must lie in a K-dimensional subspace, namely the column space of $\mathbf{U}$.

Recap: Linear Autoencoders and PCA

- Review from CSC2515: linear autoencoders with squared error loss are equivalent to Principal Component Analysis (PCA).
- Two equivalent formulations:
  - Find the subspace that minimizes the reconstruction error.
  - Find the subspace that maximizes the projected variance.
- The optimal subspace is spanned by the dominant eigenvectors of the empirical covariance matrix.



"Eigenfaces"

Example: Regularized Linear Autoencoders

- For simplicity, assume the inputs $\mathbf{x}^{(i)}$ are already centered (zero-mean).
- Encoder $\mathbf{z} = f_{\text{enc}}(\mathbf{x}) = \mathbf{W}_1\mathbf{x}$ and decoder $\hat{\mathbf{x}} = f_{\text{dec}}(\mathbf{z}) = \mathbf{W}_2\mathbf{z}$
- Squared error cost function:

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

- Previous argument shows that one optimal solution is $\mathbf{W}_1 = \mathbf{U}^\top$ and $\mathbf{W}_2 = \mathbf{U}$, where columns of $\mathbf{U}$ are the top K principal components
- But there's a symmetry: for any invertible matrix $\mathbf{A}$, we can transform the solution as:

$$\mathcal{T}_{\mathbf{A}}(\mathbf{W}_1, \mathbf{W}_2) = (\mathbf{A}\mathbf{W}_1, \mathbf{W}_2\mathbf{A}^{-1})$$

- Hence, we can only identify the principal subspace, not the individual principal components.

Example: Regularized Linear Autoencoders

- We can break the symmetry by adding a non-uniform ℓ_2 regularizer which penalizes some columns more heavily than others:

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

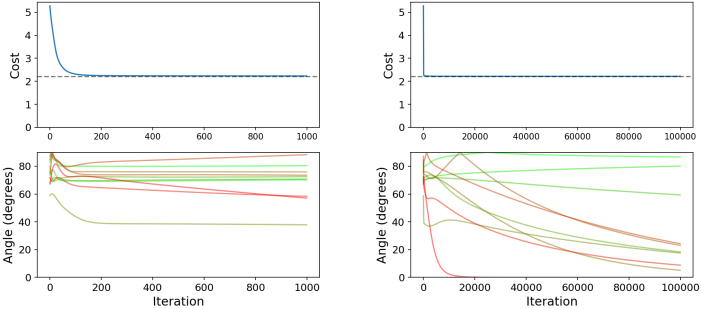
  where $\mathbf{\Lambda}$ is a diagonal matrix with increasing diagonal entries.

- Intuition: want to allocate higher-variance directions to columns with smaller penalties.
- Optimal solution:

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

Example: Regularized Linear Autoencoders

- What happens when we try to optimize this using gradient descent?
  - JAX code given in the course readings
- We measure the angle between each column of $\mathbf{W}_1$ and the corresponding principal component.



Example: Regularized Linear Autoencoders

- Can we explain this using the Hessian at the global optimum?
- **Hypothesis:** rotation of the latent space corresponds to a direction of low curvature.
- **Recall:** we can measure the curvature in a direction $\mathbf{v}$ using the Rayleigh quotient $\mathbf{v}^\top \mathbf{H}\mathbf{v} / \|\mathbf{v}\|^2$.

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

- It's a high dimensional space, so there are lots of directions we can look at. How to choose?

Example: Regularized Linear Autoencoders

- Rescaling all the weights has a big effect on the reconstruction error.
- Transformation group:

$$\mathcal{T}_\gamma(\mathbf{W}_1, \mathbf{W}_2) = (\gamma \mathbf{W}_1, \gamma \mathbf{W}_2)$$

- Let $\mathbf{v}$ be the directional derivative with respect to this transformation group at $\gamma = 1$.

```
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))
```

Output: `1.3586808`

Example: Regularized Linear Autoencoders

- Rotating the latent space doesn't affect the reconstruction error, and has a subtle effect on the regularizer.
- Transformation group:

$$\mathcal{T}_\theta(\mathbf{W}_1, \mathbf{W}_2) = (\mathbf{Q}_\theta \mathbf{W}_1, \mathbf{W}_2 \mathbf{Q}_\theta^\top),$$

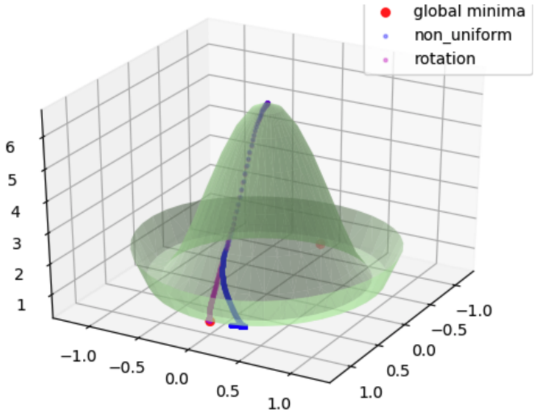
where $\mathbf{Q}_\theta$ is a Givens rotation matrix which rotates the first 2 columns by θ radians. Compute the directional derivate at $\theta = 0$.

```
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))
```

Output: `0.00041926137`

Example: Regularized Linear Autoencoders

- So the curvature in the "rotation direction" is about 3000 times smaller than the curvature in the "scaling direction"!
- Visualization of the cost landscape (Bao et al., 2020):



Gauss-Newton Hessian

Gauss-Newton Hessian

- Some problems with the Hessian
  - Not necessarily PSD
    - Newton's method can get stuck at saddle points (Lecture 4)
    - Solving linear systems with conjugate gradient requires PSD (later today)
  - Requires second derivatives of the activation function (problematic for ReLU, etc.)
- The Gauss-Newton Hessian is an approximation which is always PSD, and is often accurate in practice

Gauss-Newton Hessian

- Let $\mathbf{z} = f(\mathbf{w}, \mathbf{x})$ denote the network's function and $\mathcal{L}$ the output space loss function
  - $\mathbf{z}$ = outputs for regression, logits for classification (important!)
  - $\mathcal{L}$ = squared error for regression, softmax-cross-entropy for classification
- Decomposition of the Hessian:

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

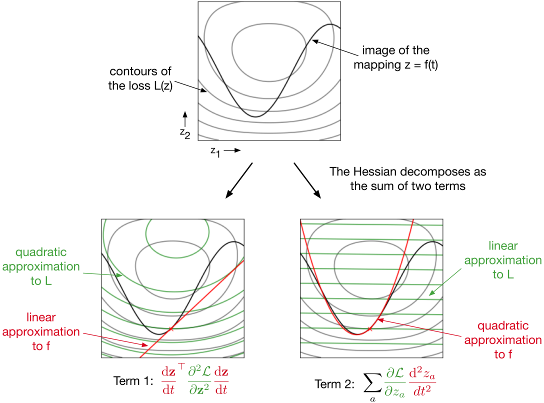
  where $\mathbf{H}_{\mathbf{z}} = \nabla^2_{\mathbf{z}} \mathcal{L}(\mathbf{z}, \mathbf{t})$ is the output Hessian.
- The Gauss-Newton Hessian, or generalized Gauss-newton (GGN) matrix, drops the second term (which empirically seems to be small in practice):

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

Gauss-Newton Hessian

- Another way to understand this approximation is that we linearize the network around the current weights:

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


contours of the loss L(z)
image of the mapping z = f(t)
z_2
z_1
The Hessian decomposes as the sum of two terms
quadratic approximation to L
linear approximation to f
linear approximation to L
quadratic approximation to f

Term 1: $\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}^{\top} \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^2} \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}$

Term 2: $\sum_a \frac{\partial \mathcal{L}}{\partial z_a} \frac{\mathrm{d}^2 z_a}{dt^2}$

This is the GGN matrix, which is always positive semidefinite.

This term may have negative eigenvalues. It vanishes if z is at the optimum.

Gauss-Newton Hessian

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

- Why $\mathbf{G}$ is PSD
  - Typical output space losses (e.g. squared error, softmax-cross-entropy) are convex, so $\mathbf{H}_{\mathbf{z}}$ is PSD
  - If $\mathbf{A}$ is a symmetric PSD matrix, then $\mathbf{B}\mathbf{A}\mathbf{B}^\top$ is symmetric and PSD for any matrix $\mathbf{B}$
- Only requires first derivatives of the network function, therefore it only requires first derivatives of ReLU
  - We're generally willing to take first derivatives of ReLU, but not second derivatives

Gauss-Newton Hessian

MVP Implementation:

$$\mathbf{G}\mathbf{v} = \mathbf{J}_{\mathbf{zw}}^{\top}\mathbf{H}_{\mathbf{z}}\mathbf{J}_{\mathbf{zw}}\mathbf{v}$$

```python
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]
```

Exercise: can you make this more efficient?

Gauss-Newton Hessian

Some gotchas:

- The term Gauss-Newton matrix is sometimes used to refer to the special case of squared error.
  - Then $\mathbf{H}_{\mathbf{z}} = \mathbf{I}$, so $\mathbf{G} = \mathbf{J}_{\mathbf{zw}} \mathbf{J}_{\mathbf{zw}}^{\top}$
  - It still makes sense to use this matrix even for other loss functions. We'll see why in Lecture 3.
- For classification, it's important to define the outputs as the logits, not the probabilities. (More insight into this in Lecture 3.)

Solving Linear Systems with Conjugate Gradient

Solving Linear Systems

- MVPs seem pretty limiting, but scientific computing has produced many powerful algorithms that exploit them.
- How to solve a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$? ($\mathbf{A} = \mathbf{H}, \mathbf{G}$, etc.)
    - **Option 1:** Construct $\mathbf{A}$ explicitly and solve the dense linear system. Only practical for small toy examples.
    - **Option 2:** (if $\mathbf{A}$ is PSD) Gradient descent on $\mathcal{J}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$
        - Only requires MVPs:

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha(\mathbf{A}\mathbf{x} - \mathbf{b})$$

        - But we need to choose α, and it converges slowly along smaller eigendirections (see Lecture 1)
    - Can we do better?
- Conjugate gradient is a powerful algorithm that uses MVPs to minimize

$$\mathcal{J}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$$

for PSD $\mathbf{A}$.

Conjugate Gradient

- Consider the Krylov subspace:

$$\mathcal{K}_k(\mathbf{A}, \mathbf{r}) = \mathrm{span}\{\mathbf{r}, \mathbf{A}\mathbf{r}, \ldots, \mathbf{A}^{k-1}\mathbf{r}\}$$

- If $\mathbf{x} \in \mathcal{K}_k(\mathbf{A}, \mathbf{b})$, then $\nabla \mathcal{J}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} \in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{b})$.
- Therefore, for any iterative algorithm initialized at $\mathbf{x} = \mathbf{0}$ which computes at most 1 gradient per iteration (e.g. GD, GD with momentum), the k^{th} iteration is contained in $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$.
- Hence,

$$\mathcal{J}(\mathbf{x}^{(k)}) \geq \min_{\mathbf{x} \in \mathcal{K}_k(\mathbf{A}, \mathbf{b})} \mathcal{J}(\mathbf{x}).$$

Conjugate Gradient

- Conjugate gradient is an iterative algorithm with the property that:

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

- Amazingly, it does this using only 1 MVP per iteration, plus cheap operations like dot products and linear combinations, with small constant factor memory overhead.
- Therefore, it achieves the optimal convergence rate among all algorithms based on MVPs and linear combinations!
- We showed in Lecture 1 that SGD requires $\mathcal{O}(\kappa)$ iterations to reach a given error. It can be shown that CG requires $\mathcal{O}(\sqrt{\kappa})$.
- For more details, see "Conjugate Gradient Without the Agonizing Pain," by Shewchuk.

Conjugate Gradient

From the user's perspective:

```
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]
```

Good idea to use 64-bit floats for numerical stability (at least for debugging):

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

Example: Sensitivity to Dataset Perturbations

Sensitivity Analysis

- Suppose we've trained a network and we want to know how the optimal weights would change if we slightly perturbed the training set.
  - E.g. influence functions: how would the predictions change if we removed data point i?
  - identifying mislabeled data
  - data poisoning attacks: attacker adds/modifies a training example so as to induce a particular misclassification
- Consider the response function, or rational reaction function

$$\mathbf{w}_\star = r(\boldsymbol{\theta}) = \arg\min_{\mathbf{w}} \mathcal{J}(\mathbf{w}; \boldsymbol{\theta})$$

- The implicit function theorem (IFT) guarantees such a function exists under certain conditions we won't worry about
- To predict the effect of a small perturbation to $\boldsymbol{\theta}$, we are intereted in the response Jacobian, or reaction Jacobian:

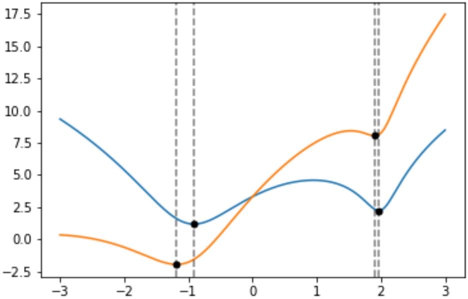
$$\mathbf{J}_{\mathbf{w}_\star \boldsymbol{\theta}} = \frac{\mathrm{d}r}{\mathrm{d}\boldsymbol{\theta}}$$

Sensitivity Analysis

- Formula for the response Jacobian:

$$\mathbf{J}_{\mathbf{w}_\star \boldsymbol{\theta}} = \frac{\mathrm{d}r}{\mathrm{d}\boldsymbol{\theta}} = -\left[\nabla^2_{\mathbf{w}} \mathcal{J}(\mathbf{w}; \boldsymbol{\theta})\right]^{-1} \nabla^2_{\mathbf{w}\boldsymbol{\theta}} \mathcal{J}(\mathbf{w}; \boldsymbol{\theta})$$

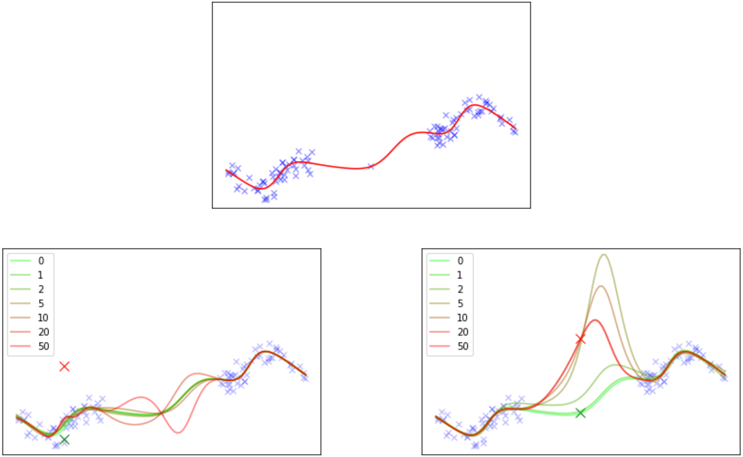
- To check that this is at least reasonable:



$\mathcal{J}(w; \lambda) = g(w) + \lambda w$ for $\lambda = 0$ and $\lambda = 3$

Sensitivity Analysis

- We can implement the formula for $\mathbf{J}_{\mathbf{w}_\star \boldsymbol{\theta}}$ just like the other examples in this lecture, solving the linear system with CG. (Full code given in the readings.)


0
1
2
5
10
20
50
0
1
2
5
10
20
50