CSC 2541: Neural Net Training Dynamics
Lecture 2 - Taylor Approximations

Roger Grosse

University of Toronto, Winter 2021
Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be differentiable at $x_0$, and $y = f(x)$.

Taylor’s Theorem implies that $f$ can be approximated by its first-order Taylor approximation, or linearization:

$$f(x) = f(x_0) + J_{yx}(x_0)(x - x_0) + o(\|x - x_0\|),$$

or

$$\Delta y = J_{yx}\Delta x + o(\|\Delta x\|).$$

$J_{yx}(x_0)$ is the Jacobian matrix of $f$ at $x_0$:

$$[J_{yx}(x_0)]_{ij} = \frac{\partial y_i}{\partial x_j} \bigg|_{x_0}$$

Typically we drop the explicit argument and just write $J_{yx}$, assuming it’s clear from context.
Examples

- **Matrix-vector product**

  \[ z = Wx \quad J_{zx} = W \]

- **Elementwise operations**

  \[ y = \exp(z) \quad J_{yz} = \begin{pmatrix} \exp(z_1) & 0 \\ \vdots & \ddots \\ 0 & \exp(z_D) \end{pmatrix} \]

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

  \[ J_{yz}v = \exp(z) \circ v \]
The gradient is an important special case.

If \( f : \mathbb{R}^m \to \mathbb{R} \), then \( J_{yx} = (\nabla y(x))^\top \). (By convention, we treat \( \nabla y(x) \) as a column vector.)

First-order Taylor approximation to a cost function \( \mathcal{J}(w) \):

\[
\mathcal{J}(w) = \mathcal{J}(w_0) + \nabla \mathcal{J}(w_0)^\top (w - w_0) + o(\|w - w_0\|)
\]

Computed using backpropagation, or reverse mode autodiff. Provided as \texttt{jax.grad}.
Jacobian Matrix

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

\[ \Delta y = R_{\Delta w} f(w) + o(\|\Delta w\|), \]

where

\[ R_{\Delta w} f(w) = \lim_{h \to 0} \frac{f(w + \Delta w) - f(w)}{h} = J_{yw} \Delta w \]

- Computed using forward mode autodiff. Provided as `jax.jvp`. (JVP = Jacobian-vector product)
The Jacobian matrix can be very large. E.g., \( J_{yw} \) 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 \( Jv \) for a vector \( v \). These are basically directional derivatives (see previous slide).
- VJPs compute \( J^\top v \). 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 \( f(x) \).

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.
- $\nabla \mathcal{J}(w) = J^\top$, 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
```
Perhaps the most elegant 3 lines of code I’ve ever seen: implementing JVP using VJP.

**Observation:** the Jacobian of the VJP function, $g(v) = J^\top v$, is just $J^\top$.

So we can compute $Jv$ by calling a VJP on the VJP!

```python
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 $w_0$ is the matrix of second derivatives:

\[
H(w_0) = \nabla^2 \mathcal{J}(w_0)
\]

\[
H_{ij} = \left. \frac{\partial^2 \mathcal{J}}{\partial w_i \partial w_j} \right|_{w = w_0}
\]

- $H$ 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 $\mathcal{J}$:

\[
\mathcal{J}(w) = \mathcal{J}(w_0) + \nabla \mathcal{J}(w_0)^\top (w - w_0) + \\
+ \frac{1}{2} (w - w_0)^\top H(w - w_0) + o(||w - w_0||^2)
\]
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 $\mathbf{v}$. 
Recall: A symmetric matrix $A$ is positive definite, written $A \succ 0$, if $v^\top Av > 0$ for any vector $v \neq 0$.

- Equivalently, all of $A$’s eigenvalues are positive.
- If the inequality isn’t necessarily strict, then $A$ is positive semidefinite (PSD), written $A \succeq 0$.

Recall: A function $f$ is convex if:

$$J(\lambda w_1 + (1-\lambda)w_0) \leq \lambda J(w_1) + (1-\lambda)J(w_0) \quad \forall w_0, w_1, \lambda \in [0, 1].$$

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

Categorizing stationary points using the spectrum of $H$

- $H$ positive definite: local minimum
- $H$ negative definite: local maximum (this is unusual)
- $H$ has positive and negative eigenvalues: saddle point (this is more common)
- $H$ is PSD but some eigenvalues are 0: could be a maximum or minimum
Close to a stationary point $\mathbf{w}_*$ we can understand the gradient dynamics using the second-order Taylor approximation

$$J(\mathbf{w}) \approx J(\mathbf{w}_*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_*)$$

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

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

$$\tilde{\mathbf{w}}^{(k)}_i = \tilde{\mathbf{w}}_i^* + (1 - \alpha \tilde{h}_i)^k (\tilde{\mathbf{w}}^{(0)}_i - \tilde{\mathbf{w}}_i^*)$$,
Gradient Descent Dynamics: Local Minimum

- Stable if $\alpha < 2\tilde{h}_{\text{max}}^{-1}$
- 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 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.
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:

```python
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.](image)

- **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
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 $x$ and predict $x$.
- 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.

\[ \mathcal{L}(x, \tilde{x}) = \|x - \tilde{x}\|^2 \]

- This network computes \( \tilde{x} = UVx \), which is a linear function.

- If \( K \geq D \), we can choose \( U \) and \( V \) such that \( UV \) is the identity. This isn’t very interesting.

- But suppose \( K < D \):
  - \( V \) maps \( x \) 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 \( U \).
Recap: Linear Autoencoders and PCA

- Review from CSC421: 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 $x^{(i)}$ are already centered (zero-mean).
- Encoder $z = f_{\text{enc}}(x) = W_1 x$ and decoder $\hat{x} = f_{\text{dec}}(z) = W_2 z$
- Squared error cost function:
  \[
  \frac{1}{2N} \sum_{i=1}^{N} \| W_2 W_1 x^{(i)} - x^{(i)} \|^2
  \]
- Previous argument shows that one optimal solution is $W_1 = U^\top$ and $W_2 = U$, where columns of $U$ are the top $K$ principal components.
- But there’s a symmetry: for any invertible matrix $A$, we can transform the solution as:
  \[
  T_A(W_1, W_2) = (AW_1, W_2 A^{-1})
  \]
- Hence, we can only identify the principal subspace, not the individual principal components.
We can break the symmetry by adding a non-uniform $\ell_2$ regularizer which penalizes some columns more heavily than others:

$$\frac{1}{2N} \sum_{i=1}^{N} \| W_2 W_1 x^{(i)} - x^{(i)} \|^2 + \frac{1}{2} \| \Lambda^{1/2} W_1 \|_F^2 + \frac{1}{2} \| W_2 \Lambda^{1/2} \|_F^2,$$

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

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

Optimal solution:

$$W_1^* = P (I - \Lambda S^{-2})^{1/2} U^T$$
$$W_2^* = U (I - \Lambda S^{-2})^{1/2} P,$$
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 $v$ using the Rayleigh quotient $v^\top Hv/\|v\|^2$.

```python
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:
  
  \[ \mathcal{T}_\gamma(W_1, W_2) = (\gamma W_1, \gamma W_2) \]

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

```python
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:

\[ \mathcal{T}_\theta(W_1, W_2) = (Q_\theta W_1, W_2 Q_\theta^\top) , \]

where \( Q_\theta \) is a Givens rotation matrix which rotates the first 2 columns by \( \theta \) radians. Compute the directional derivate at \( \theta = 0 \).

```python
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
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 $z = f(w, x)$ denote the network’s function and $\mathcal{L}$ the output space loss function
  - $z =$ outputs for regression, logits for classification (important!)
  - $\mathcal{L} =$ squared error for regression, softmax-cross-entropy for classification
- Decomposition of the Hessian:
  \[
  \nabla^2 \mathcal{J}_{x,t}(w) = J_{zw}^{\top} H_z J_{zw} + \sum_a \frac{\partial \mathcal{L}}{\partial z_a} \nabla^2_w [f(x, w)]_a,
  \]
  where $H_z = \nabla^2_z \mathcal{L}(z, t)$ 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):
  \[
  G = J_{zw}^{\top} H_z J_{zw}
  \]
Another way to understand this approximation is that we linearize the network around the current weights:

\[ f_{\text{lin}}(w', x) = f(w, x) + J_yw(w' - w) \]
Gauss-Newton Hessian

\[ G = J_{zw}^\top H_z J_{zw} \]

- Why \( G \) is PSD
  - Typical output space losses (e.g. squared error, softmax-cross-entropy) are convex, so \( H_z \) is PSD
  - If \( A \) is a symmetric PSD matrix, then \( BAB^\top \) is symmetric and PSD for any matrix \( B \)
- 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:

\[ Gv = J_{zw}^\top H_z J_{zw} v \]

```python
def gnhvp(f, L, w, v):
    y, 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?
Some gotchas:

- The term Gauss-Newton matrix is sometimes used to refer to the special case of squared error.
  - Then $H_z = I$, so $G = J_{zw}J_{zw}^T$
  - 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 $Ax = b$? ($A = H, G, \text{ etc.}$)
  - **Option 1:** Construct $A$ explicitly and solve the dense linear system. Only practical for small toy examples.
  - **Option 2:** (if $A$ is PSD) Gradient descent on $J(x) = \frac{1}{2} x^\top Ax - b^\top x$
    - Only requires MVPs:
      $$x^{(k+1)} \leftarrow x^{(k)} - \alpha (Ax - b)$$
    - But we need to choose $\alpha$, 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
  $$J(x) = \frac{1}{2} x^\top Ax - b^\top x$$
  for PSD $A$. 
Consider the Krylov subspace:

\[ \mathcal{K}_k(A, r) = \text{span}\{r, Ar, \ldots, A^{k-1}r\} \]

If \( x \in \mathcal{K}_k(A, b) \), then \( \nabla J(x) = Ax - b \in \mathcal{K}_{k+1}(A, b) \).

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

Hence,

\[ J(x^{(k)}) \geq \min_{x \in \mathcal{K}_k(A, b)} J(x). \]
Conjugate Gradient

- Conjugate gradient is an iterative algorithm with the property that:
  \[ x^{(k)} = \arg \min_{x \in \mathcal{K}_k(A,b)} J(x). \]

- 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 \( O(\kappa) \) iterations to reach a given error. It can be shown that CG requires \( O(\sqrt{\kappa}) \).

- For more details, see “Conjugate Gradient Without the Agonizing Pain,” by Shewchuk.
Conjugate Gradient

From the user’s perspective:

```python
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):

```python
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
  $$ w_\star = r(\theta) = \arg \min_w J(w; \theta) $$
- 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 $\theta$, we are interested in the response Jacobian, or reaction Jacobian:
  $$ J_{w\star\theta} = \frac{dr}{d\theta} $$
Sensitivity Analysis

- **Formula for the response Jacobian:**

  \[
  J_{w*\theta} = \frac{dr}{d\theta} = - \left[ \nabla_w^2 \mathcal{J}(w; \theta) \right]^{-1} \nabla_{w\theta} \mathcal{J}(w; \theta)
  \]

- **To check that this is at least reasonable:**

  \[\mathcal{J}(w; \lambda) = g(w) + \lambda w \text{ for } \lambda = 0 \text{ and } \lambda = 3\]
Sensitivity Analysis

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