Optimization by Following Gradients

- Fundamentally, we’re interested in machines that we train by optimizing parameters
- How do we select these parameters?
- In differentiable programming, we often define an objective function that we *minimize* (or *maximize*) with respect to *(w.r.t.)* these parameters
- That is, we’re looking for points at which the *gradient of the objective function is zero w.r.t. the parameters*
Optimization by Following Gradients

- Gradient based optimization is a big field.
  - First order methods, second order methods, subgradient methods...
- With Differentiable Programming, we’re primarily interested in the first-order methods\(^1\).
  - Primarily using variants of gradient descent: a function \( F(x) \) has a minima\(^2\) (or a saddle-point) at a point \( x = a \) where \( a \) is given by applying \( a_{n+1} = a_n - \alpha \nabla F(a_n) \) until convergence from some initial point \( a_0 \)

\(^{1}\)Second-order gradient optimizers are potentially better, but for systems with many variables are currently impractical as they require computing the Hessian.

\(^{2}\)not necessarily global or unique

What Are Gradients?

- The derivative in 1D \( \frac{\Delta y}{\Delta x} \)
- The gradient of a straight line is \( \Delta y / \Delta x \)
- For an arbitrary real-valued function, \( f(a) \), we can approximate the derivative, \( f'(a) \) using the gradient of the secant line defined by \((a, f(a))\) and a point a small distance, \( h \), away \((a + h, f(a + h))\): \( f'(a) \approx \frac{f(a + h) - f(a)}{h} \)
- This expression is ‘Newton’s Difference Quotient’
- As \( h \) becomes smaller, the approximated derivative becomes more accurate. Take the limit as \( h \to 0 \), we have

\[
\frac{df}{da} = f'(a) = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h}
\]
What Are Gradients?

The derivative of $y = x^2$ from first principles

$$y = x^2$$

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{(x + h)^2 - x^2}{h}$$

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{x^2 + h^2 + 2hx - x^2}{h}$$

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{h^2 + 2hx}{h}$$

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{h(2x)}{h}$$

$$\frac{dy}{dx} = 2x$$

Numerical Approximation of Derivatives

- For numerical computation of derivatives it is better to use a “centralized” definition:

$$f'(a) = \lim_{h \to 0} \frac{f(a+h)-f(a-h)}{2h}$$

- The bit inside the limit is known as the ”symmetric difference quotient”

- For small values of $h$, this has less error than the standard one-sided difference quotient
What Are Gradients?

- If you are going to use difference quotients to estimate derivatives you need to be aware of potential rounding errors due to floating point representations.

- Calculating derivatives this way using less than 64-bit precision is rarely going to be useful. (Numbers are not represented exactly, so even if \( h \) is represented exactly, \( x + h \) will probably not be.)

- You need to pick an appropriate \( h \) – too small and the subtraction will have a large rounding error!

What Are Gradients?

- Deep learning is all about optimizing deeper functions; functions that are compositions of other functions:

\[
e.g. \quad z = f \circ g (x) = f (g (x))
\]

- The chain rule of calculus tells us how to differentiate compositions of functions:

\[
\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}
\]
What Are Gradients?

Or, derive from the first principle:

For a vector function, \( y(t) \), this can be split into its constituent coordinate functions:

\[ y(t) = (y_1(t), \ldots, y_n(t)) \]

The derivative is a (tangent) vector:

\[ y'(t) = (y_1'(t), \ldots, y_n'(t)) \]

which consists of the derivatives of the coordinate functions.

Equivalently, if the limit exists, then

\[ y'(t) = \lim_{h \to 0} \frac{y(t+h) - y(t)}{h} \]
Functions of Multiple Variables: Partial Differentiation

- What if the function we’re trying to deal with has multiple variables\(^3\) (e.g. \(f(x, y) = x^2 + xy + y^2\))?
  - This expression has a pair of partial derivatives, \(\frac{\partial f}{\partial x} = 2x + y\) and \(\frac{\partial f}{\partial y} = x + 2y\), computed by differentiating with respect to each variable \(x\) and \(y\) whilst holding the other(s) constant.

- Generally partial derivative of a function \(f(x_1, \ldots, x_n)\) at a point \((a_1, \ldots, a_n)\) is given by:
  \[
  \frac{\partial f}{\partial x_i}(a_1, \ldots, a_n) = \lim_{h \to 0} \frac{f(a_1, \ldots, a_i + h, \ldots, a_n) - f(a_1, \ldots, a_i, \ldots, a_n)}{h}.
  \]

- The vector of partial derivatives of a scalar-value multivariate function, \(f(x_1, \ldots, x_n)\) at a point \((a_1, \ldots, a_n)\), can be arranged into a vector, gradient of \(f \at \mathbf{a}\).
  \[\nabla f(a_1, \ldots, a_n) = \frac{\partial f}{\partial x_1}(a_1, \ldots, a_n), \ldots, \frac{\partial f}{\partial x_n}(a_1, \ldots, a_n)\]

- For a vector-valued multivariate functions, the partial derivatives form a matrix is called the Jacobian.

Functions of Vectors and Matrices: Partial Differentiation

For the kinds of functions (and programs) that we’ll look at optimizing in this course have a number of typical properties:

- They are scalar-valued
- We’ll look at programs with multiple losses, but ultimately we can just consider optimizing with respect to the sum of the losses.
- They involve multiple variables, which are often wrapped up in the form of vectors or matrices, and more generally tensors.
- How will we find the gradients of these
Chain Rule for Vectors

- Suppose that \( x \in \mathbb{R}^m \), \( y \in \mathbb{R}^n \), \( g \) maps from \( \mathbb{R}^m \) to \( \mathbb{R}^n \) and \( f \) maps from \( \mathbb{R}^n \) to \( \mathbb{R} \).
- If \( y = g(x) \) and \( z = f(y) \), then
  \[
  \frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}
  \]
- Equivalently, in vector notation:
  \[
  \nabla_x z = \left( \frac{\partial y}{\partial x} \right)^T \nabla_y z
  \]
  here \( \frac{\partial y}{\partial x} \) is the nxm Jacobian matrixs

Chain Rule for Tensors

- Conceptually, the simplest way to think about gradients of tensors is to imagine flattening them into vectors, computing the vector-valued gradient and then reshaping the gradient back into a tensor.
- In this way we’re still just multiplying Jacobians by gradients. More formally, consider gradient of a scalar \( z \) with respect to a tensor \( X \) to be denoted as \( \nabla_X z \).
- Indices into \( X \) now have multiple coordinates, but we can generalize by using a single variable \( i \) to represent the complete tuple of indices.
- For all index tuples \( i \), \( (\nabla_X z)_i \) gives \( \frac{\partial z}{\partial X_i} \)
- Thus, if \( Y = g(X) \) and \( z = f(Y) \) then
  \[
  \nabla_X z = \sum_j (\nabla_X Y_j) \frac{\partial z}{\partial Y_j}
  \]
Example: $\nabla_W f(XW)$

Let $D = XW$ where the rows of $X \in \mathbb{R}^{n \times m}$ contain some fixed features, and $W \in \mathbb{R}^{m \times h}$ is a matrix of weights.

Also let $L = f(D)$ be some scalar function of $D$ that we wish to minimize.

What are the derivatives of $L$ with respect to the weights $W$?

- Start by considering a specific weight, $W_{uv}$: $\frac{\partial L}{\partial W_{uv}} = \sum_{i,j} \frac{\partial L}{\partial D_{ij}} \frac{\partial D_{ij}}{\partial W_{uv}}$.
- We know that $\frac{\partial D_{ij}}{\partial W_{uv}} = 0$ if $j \neq v$ because $D_{ij}$ is the dot product of row $i$ of $X$ and column $j$ of $W$.
- Therefore, we can simplify the summation to only consider cases where $j = v$: $\sum_{i,j} \frac{\partial L}{\partial D_{ij}} \frac{\partial D_{ij}}{\partial W_{uv}} = \sum_i \frac{\partial L}{\partial D_{iv}} \frac{\partial D_{iv}}{\partial W_{uv}}$.
- What is $\frac{\partial D_{iv}}{\partial W_{uv}}$?

$$D_{iv} = \sum_{k=1}^{q} X_{ik} W_{kv}$$

$$\frac{\partial D_{iv}}{\partial W_{uv}} = \frac{\partial}{\partial W_{uv}} \sum_{k=1}^{q} X_{ik} W_{kv} = \sum_{k=1}^{q} \frac{\partial}{\partial W_{uv}} X_{ik} W_{kv}$$

$$\therefore \frac{\partial D_{iv}}{\partial W_{uv}} = X_{iu}$$
Example: $\nabla_W f (XW)$

Putting every together, we have:

$$\frac{\partial L}{\partial W_{uv}} = \sum_i \frac{\partial L}{\partial D_{iv}} X_{iu}$$

As we’re summing over multiplications of scalars, we can change the order:

$$\frac{\partial L}{\partial W_{uv}} = \sum_i X_{iu} \frac{\partial L}{\partial D_{iv}}.$$ 

and note that the sum over $i$ is doing a dot product with row $u$ and column $v$ if we transpose $X_{iu}$ to $X^T u_i$:

$$\frac{\partial L}{\partial W_{uv}} = \sum_i X^T_{ui} \frac{\partial L}{\partial D_{iv}}.$$ 

We can then see that if we want this for all values of $W$ it simply generalizes to:

$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial D}.$$ 

---

What Does a Gradient Do?

- In your early calculus lessons you likely had it hammered into you that gradients represent rates of change of functions.
- This is of course totally true...
- But, it isn’t a particularly useful way to think about the gradients of a loss with respect to the weights of a parameterized function.
- The gradient of the loss with respect to a parameter tells you how much the loss will change with a small perturbation to that parameter.
Singular Value Decomposition

- Let’s now change direction and look at using some differentiation and Singular Value Decomposition (SVD).
- For complex $A$:
  \[ A = U \Sigma V^* \]
  where $V^*$ is the conjugate transpose of $V$

For real $A$:
\[ A = U \Sigma V^T \]

SVD has many uses:
- Computing the Eigendecomposition:
  Eigenvectors of $AA^T$ are columns of $U$, Eigenvectors of $A^TA$ are columns of $V$,
  and the non-zero values of $\Sigma$ are the square roots of the non-zero eigenvalues of both $AA^T$ and $A^TA$.
- Dimensionality reduction
  ...use to compute PCA
- Computing the Moore-Penrose Pseudoinverse
  for real $A$: $A^+ = V \Sigma^* U^T$ where $\Sigma^*$ is formed by taking the reciprocal of every non-zero diagonal element and transposing the result.
- Low-rank approximation and matrix completion
  if you take the $\rho$ columns of $U$, and the $\rho$ rows of $V^T$ corresponding to the $\rho$ largest singular values, you can form the matrix $A_\rho = U_\rho \Sigma_\rho V_\rho^T$
  which will be the best rank-$\rho$ approximation of the original $A$ in terms of the Frobenius norm.
Computing SVD using Gradients

- There are many standard ways of computing the SVD:
  - e.g. ‘Power iteration’, or ‘Arnoldi iteration’ or ‘Lanczos algorithm’ coupled with the ‘Gram-Schmidt process’ for orthonormalization
- but, these don’t necessarily scale up to really big problems
  - e.g. computing the SVD of a sparse matrix with 17770 rows, 480189 columns and 100480507 non-zero entries!
  - this corresponds to the data provided by Netflix when they launched the Netflix Challenge in 2006.
- OK, so what can you do?
  - The ‘Simon Funk’ solution: realise that there is a really simple (and quick) way to compute the SVD by following gradients...

One of the definitions of rank- SVD of a matrix $A$ is that it minimises reconstruction error in terms of the Frobenius norm

Without loss of generality we can write SVD as a 2-matrix decomposition $A = ^*U^*V^T$ by rolling in the square roots of $\Sigma$ to both $^*U$ and $^*V$:

$$\hat{U} = U\Sigma^{0.5} \quad \text{and} \quad \hat{V}^T = \Sigma^{0.5} V^T.$$  

Then we can define the decomposition as finding:

$$\min_{\hat{U}, \hat{V}} (\|A - \hat{U}\hat{V}^T\|_F^2)$$
Deriving a gradient-descent solution to SVD

Start by expanding our optimisation problem:

\[
\min_{\hat{U}, \hat{V}}(\|A - \hat{U}\hat{V}^T\|_F^2) = \min_{\hat{U}, \hat{V}}(\sum_r \sum_c (A_{rc} - \hat{U}_r \hat{V}_c)^2)
\]

\[
= \min_{\hat{U}, \hat{V}}(\sum_r \sum_c (A_{rc} - \sum_{\rho=0}^\rho \hat{U}_{rp} \hat{V}_{cp})^2)
\]

Let \(e_{rc} = A_{rc} - \sum_{\rho=0}^\rho \hat{U}_{rp} \hat{V}_{cp}\) denote the error. Then, our problem becomes:

\[
\text{Minimise } J = \sum_r \sum_c e_{rc}^2
\]

We can then differentiate with respect to specific variables \(\hat{U}_{rq}\) and \(\hat{V}_{cq}\)