# Course Notes for EE227C (Spring 2018): Convex Optimization and Approximation

Instructor: Moritz Hardt

Email: hardt+ee227c@berkeley.edu

Graduate Instructor: Max Simchowitz

Email: msimchow+ee227c@berkeley.edu

April 26, 2018

## 16 Backpropagation and adjoints

From now onward, we give up the luxuries afforded by convexity and move to the realm of non-convex functions. In the problems we have seen so far, deriving a closed-form expression for the gradients was fairly straightforward. But, doing so can be a challenging task for general non-convex functions. In this class, we are going to focus our attention on functions that can be expressed as compositions of multiple functions. In what follows, we will introduce *backpropagation* - a popular technique that exploits the composite nature of the functions to compute gradients incrementally.

### 16.1 Warming up

A common view of backpropagation is that "*it's just chain rule*". This view is not particular helpful, however, and we will see that there is more to it. As a warm-up example, let us look at the following optimization problem with $g : \mathbb{R}^n \to \mathbb{R}$, $f : \mathbb{R} \to \mathbb{R}$:

$$\min_x f(g(x)). \tag{1}$$

Using a similar trick to the one we saw in the context of ADMM, we can rewrite this problem as

$$\begin{aligned} &\min_x f(z) \\ \text{s.t. } &z = g(x) \end{aligned} \tag{2}$$

Note that we have converted our original unconstrained problem in $x$ into a constrained optimization problem in $x$ and $z$ with the following Lagrangian,

$$\mathcal{L}(x, z, \lambda) = f(z) + \lambda(g(x) - z). \tag{3}$$

Setting $\nabla\mathcal{L} = 0$, we have the following optimality conditions,

$$0 = \nabla_x \mathcal{L} = \lambda g'(x) \Leftrightarrow 0 = \lambda g'(x) \tag{4a}$$
$$0 = \nabla_z \mathcal{L} = f'(z) - \lambda \Leftrightarrow \lambda = f'(z) \tag{4b}$$
$$0 = \nabla_\lambda \mathcal{L} = g(x) - z \Leftrightarrow z = g(x) \tag{4c}$$

which implies

$$\begin{aligned} 0 &= f'(g(x))g'(x) \\ &= \nabla_x f(g(x)) \qquad \text{(by chain rule)} \end{aligned}$$

Hence, solving the Lagrangian equations gave us an incremental way of computing gradients. As we will see shortly, this holds at great generality. It is important to notice that we did *not* use the chain rule when solving the equations in (4). The chain rule only showed up in the proof of correctness.

## 16.2 General formulation

Any composite function can be described in terms of its computation graph. As long as the elementary functions of the computation graph are differentiable, we can perform the same procedure as above. Before moving ahead, let us introduce some notation:

- Directed, acyclic computation graph: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- Number of vertices: $|V| = n$
- Set of ancestors of $i^{th}$ node: $\alpha(i) = \{j \in \mathcal{V} : (j, i) \in \mathcal{E}\}$
- Set of successors of $i^{th}$ node: $\beta(i) = \{j \in \mathcal{V} : (i, j) \in \mathcal{E}\}$
- Computation at the $i^{th}$ node: $f_i(z_{\alpha(i)}), \quad f_i : \mathbb{R}^{|\alpha(i)|} \to \mathbb{R}^{|\beta(i)|}$
- Nodes:
  - Input - $z_1, \ldots, z_d$
  - Intermediate - $z_{d+1}, \ldots, z_{n-1}$
  - Output - $z_n$

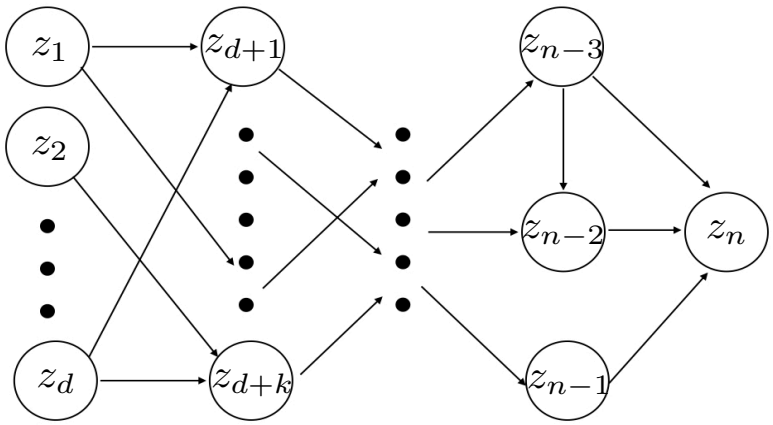


Figure 1: Computation graph

Then, the general formulation is given by

$$\begin{aligned} &\min \;\; z_n \\ &\text{s.t.} \;\; z_i = f_i(z_{\alpha(i)}). \end{aligned} \tag{5}$$

with the following Lagrangian,

$$\mathcal{L}(x, z, \lambda) = z_n - \sum_i \lambda_i (z_i - f_i(z_{\alpha(i)})). \tag{6}$$

As in the warm-up example, we set $\nabla \mathcal{L} = 0$. This can be viewed as an algorithm comprising two separate steps:

**Backpropagation algorithm**

- *Step 1*: Set $\nabla_\lambda \mathcal{L} = 0$, i.e.,

$$\nabla_{\lambda_i} \mathcal{L} = z_i - f_i(z_{\alpha(i)}) = 0 \Leftrightarrow z_i = f_i(z_{\alpha(i)}) \tag{7}$$

  *Observation*: This is known as *forward pass or forward propagation* as the values at the nodes ($z_i$) are computed using the values of the ancestors.

- *Step 2*: Set $\nabla_{z_j} \mathcal{L} = 0$,
  - for $j = n$,

$$\begin{aligned} 0 = \nabla_{z_j} \mathcal{L} &= 1 - \lambda_n \\ \Leftrightarrow \lambda_n &= 1 \end{aligned}$$

– for $j < n$,

$$\begin{aligned} 0 &= \nabla_{z_j} \mathcal{L} \\ &= \nabla_{z_j} (z_n - \sum_i \lambda_i (z_i - f_i(z_{\alpha(i)}))) \\ &= -\sum_i \lambda_i (\nabla_{z_j}[z_i] - \nabla_{z_j} f_i(z_{\alpha(i)})) \\ &= -\lambda_j + \sum_i \lambda_i \nabla_{z_j} f_i(z_{\alpha(i)}) \\ &= -\lambda_j + \sum_{i \in \beta(j)} \lambda_i \frac{\partial f_i(z_{\alpha(i)})}{\partial z_j} \\ \Leftrightarrow \quad \lambda_j &= \sum_{i \in \beta(j)} \lambda_i \frac{\partial f_i(z_{\alpha(i)})}{\partial z_j} \end{aligned}$$

*Observation*: This is known as *backward pass or backpropagation* as $\lambda_i's$ are computed using the gradients and values of $\lambda$ at successor nodes in the computation graph.

## 16.3 Connection to chain rule

In this section, we will prove a theorem that explains why backpropagation allows us to calculate gradients incrementally.

**Theorem 16.1.** *For all $1 \leqslant j \leqslant n$, we have*

$$\lambda_j = \frac{\partial f(x)}{\partial z_j},$$

*i.e., the partial derivative of the function $f$ at $x$ w.r.t to the $j^{th}$ node in the graph.*

*Proof.* We assume for simplicity that the computation graph has $L$ layers and edges exist only between consecutive layers, i.e., $f = f_L \circ \cdots \circ f_1$. The proof is by induction over layers (starting from the output layer).
*Base case*: $\lambda_n = 1 = \frac{\partial f_n(x)}{\partial z_n} = \frac{\partial z_n}{z_n}$.

*Induction*: Fix $p^{th}$ layer and assume claim holds for nodes in all subsequent layers $l > p$.

Then, for node $z_j$ in layer $p$,

$$\begin{aligned} \lambda_j &= \sum_{i \in \beta(j)} \lambda_i \frac{\partial f_i(z_{\alpha(i)})}{\partial z_j} \\ &= \sum_{i \in \beta(j)} \frac{\partial f(x)}{\partial z_i} \frac{\partial z_i}{z_j} \qquad (z_{\beta(j)} \text{ belong to layer } p+1) \\ &= \frac{\partial f(x)}{\partial z_j} \qquad \text{(by multivariate chain rule).} \end{aligned}$$

■

(∗) Note that the proof for arbitrary computation graphs is by induction over the partial order induced by the reverse computation graph.

**Remarks**

1. Assuming elementary node operations cost constant time, cost of both the forward and backward pass is $O(|\mathcal{V}| + |\mathcal{E}|) \Rightarrow$ Linear time!
2. Notice that the algorithm itself does not use the chain rule, only its correctness guarantee uses it.
3. Algorithm is equivalent to the "*method of adjoints*" from control theory introduced in the 60's. It was rediscovered by Baur and Strassen in '83 for computing partial derivatives [BS83]. More recently, it has received a lot of attention due to its adoption by the Deep Learning community since the 90's.
4. This algorithm is also known as *automatic differentiation*, not to be confused with
   (a) Symbolic differentiation
   (b) Numerical differentiation

## 16.4 Working out an example

Suppose we have a batch of data $X \in \mathbb{R}^{n \times d}$ with labels $y \in \mathbb{R}^n$. Consider a two-layer neural net given by weights $W_1, W_2 \in \mathbb{R}^{d \times d}$ :

$$f(W_1, W_2) = \|\sigma(XW_1)W_2 - y\|^2$$

To compute gradients, we only need to implement forward/backward pass for the elementary operations:

- Squared norm
- Subtraction/addition
- Component-wise non-linearity $\sigma$
- Matrix multiplication

Observe that the partial derivatives for the first three operations are easy to compute. Hence, it suffices to focus on matrix multiplication.

**Back-propagation for matrix completion**

The two steps of the backpropagation algorithm in this context are:

**Forward Pass:**

- Input: $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times d}$
- Output: $C = AB \in \mathbb{R}^{m \times d}$

**Backward Pass:**

- Input: Partial derivatives $\Lambda \in \mathbb{R}^{m \times d}$ (also $A, B, C$ from forward pass)
- Output:
  - $\Lambda_1 \in \mathbb{R}^{m \times n}$ (partial derivatives for left input)
  - $\Lambda_2 \in \mathbb{R}^{n \times d}$ (partial derivatives for right input)

**Claim 16.2.** $\Lambda_1 = \Lambda B^T, \Lambda_2 = A^T \Lambda$

*Proof.*

$$f = \sum_{i,j} \lambda_{ij} C_{ij} = \sum_{i,j} (AB)_{ij} = \sum_{i,j} \lambda_{ij} \Sigma_k a_{ik} b_{kj}.$$

So, by Lagrange update rule,

$$(\Lambda_1)_{pq} = \frac{\partial f}{\partial a_{pq}} = \sum_{i,j,k} \lambda_{ij} \frac{\partial a_{ik}}{\partial a_{pq}} b_{kj} = \sum_j \lambda_{pj} b_{qj} = (\Lambda B^T)_{pq}.$$

Using the same approach for partial derivative w.r.t. $B$, we get

$$(\Lambda_2)_{pq} = (A^T \Lambda)_{pq}$$

■

# References


[BS83] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical computer science*, 22(3):317–330, 1983.