Deep Learning 236606
Lecture 4
SGD generalizes in convex-Lipschitz problems
Multi unit/layer neural networks
Computation Graph
Backpropagation
Agenda

Last lecture:
- The binary neuron and its expressive power
- Activation functions
- Logistic regression
- Gradient descent, Stochastic gradient descent
- In Tutorial: Total differential, matrix calculus, chain rule

Today:
- Generalization bound for SGD of convex-Lipschitz problems
- Neural networks
- Computation graphs
- Training NNs: backpropagation
Recap:

- SGD is like GD, but instead of optimizing over entire training set, perform each gradient step over one random example

Stochastic gradient descent (SGD)

- Initialize $\theta_1$, choose learning rate $\eta$
- Iterate $t$ until convergence ($t = T$)
  - Choose at random a labeled example $(x, y) \sim D$
  
  $\theta_{t+1} = \theta_t - \eta \nabla \ell(\theta_t, x, y)$

- Output final hypothesis $\theta_T$

- **Computational** advantage over GD per iteration: $O(d)$ instead of $O(dm)$
SGD and GD in action

A = SGD

SGD

GD
SGD example: perceptron

Recall perceptron decision: $\hat{y}_t = \text{sign}(w_t \cdot x_t)$

- Loss function:
  $$\ell(w, x_t, y_t) = -y_t w \cdot x_t \mathbb{I}(\hat{y}_t \neq y_t)$$

- $\nabla \ell = -y_t x_t \mathbb{I}(\hat{y}_t \neq y_t)$

- SGD step (set $\eta = 1$):
  $$w_{t+1} = w_t - \eta \nabla \ell = w_t + y_t x_t \mathbb{I}(\hat{y}_t \neq y_t)$$
Mini-batch SGD and other variants

- **Mini-batch**: Instead of one training example, work with a small batch of examples (e.g. 64)
- Many SGD **variants**: learning rate schedules, memory (momentum), hypothesis averaging, variance reduction
- We will see some of these later on
We will prove an SGD generalization guarantee (and GD convergence guarantee) for a large class of convex, Lipschitz learning problems.

Let’s first make sure we recall:

- Jensen inequality
- Lipschitzness
Jensen inequality

Let $f$ be convex, $x_i$ reals and $a_i$ positive reals. Then

$$f \left( \frac{\sum_i a_i x_i}{\sum_i a_i} \right) \leq \frac{\sum_i a_i f(x_i)}{\sum_i a_i}$$

if $X$ is a random variable

$$f \left( E[X] \right) \leq E f(X)$$
Lipschitz function

\[ f : \mathbb{R}^d \rightarrow \mathbb{R}^k \] is \( \alpha \)-Lipschitz over \( S \subseteq \mathbb{R}^d \) if for all \( u, v \in S \),

\[ ||f(u) - f(v)|| \leq \alpha ||u - v|| \]

Example 1:

- The mean value theorem states that for \( f : \mathbb{R} \rightarrow \mathbb{R} \) continuous on \([a, b]\) and differentiable in \((a, b)\), there exists a point \( c \in (a, b) \),

\[ f'(c) = \frac{f(b) - f(a)}{b - a} \]

in other words,

\[ f(b) - f(a) = f'(c)(b - a) \]

Result: \( f \) is Lipschitz over \( S \) iff \( f'(c) \) is bounded in \( S \).
Example 2: The function 

$$f(x) = \log(1 + e^x)$$

is 1-Lipschitz over $\mathbb{R}$
We will analyze an SGD variant called **Averaged-SGD**. Its final hypothesis is the average of all hypotheses generated in the process, Denote this averaged hypothesis by $\bar{w} = \sum_t w_t$.

- We consider convex learning problems that are $B$-bounded: $\mathcal{H} = \{ w : \|w\| < B \}$
- And the loss function $\ell(w, x, y)$ is $\alpha$-Lipschitz.

**A-SGD guarantee**

Let $(\mathcal{H}, \mathcal{X}, \mathcal{Y}, \ell)$ be a convex $B$-bounded learning problem where $\ell$ is $\alpha$-Lipschitz. Let $w^*$ be a minimizer of the true loss $L_D(w)$. If $\bar{w}$ is the output of the Averaged-SGD algorithm, then

$$E\{L_D(\bar{w})\} - L_D(w^*) \leq \frac{B\alpha}{\sqrt{T}}$$
Main idea for SGD bound

\[ L_0 = \mathbb{E}_{x,y \sim D} \ell(y, h_{w_t}(x)) \]

\[ \implies L_0 = w_t - \eta \mathbb{E}_{x,y \sim D} \nabla \ell(y, h_{w_t}(x)) \]

\[ w_{t+1} = w_t - \eta \nabla L_0 \]
Lemma 1: Let \( \mathbf{v}_1, \ldots, \mathbf{v}_T \) be any vectors such that \( ||\mathbf{v}_t|| \leq \alpha \). Consider the update rule: \( \mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{v}_t \). For any \( \mathbf{w}^* \), \( ||\mathbf{w}^*|| \leq B \), and \( \eta = B/(\alpha \sqrt{T}) \),

\[
\sum_{t=1}^{T} (\mathbf{w}_t - \mathbf{w}^*) \cdot \mathbf{v}_t \leq B \alpha \sqrt{T}
\]

Proof:

\[
(\mathbf{w}_t - \mathbf{w}^*) \cdot \mathbf{v}_t = \frac{1}{2\eta} \left( -||\mathbf{w}_t - \mathbf{w}^* - \eta \mathbf{v}_t||^2 + ||\mathbf{w}_t - \mathbf{w}^*||^2 + \eta^2 ||\mathbf{v}_t||^2 \right)
\]

- \( ||a - b||^2 = -||a - b||^2 + 2\eta \mathbf{v}_t \cdot (\mathbf{w}_t - \mathbf{w}^*) - \eta^2 ||\mathbf{v}_t||^2 \)
- \( 2\eta > 1/\eta \)
\[
(w_t - w^*) \cdot v_t = \frac{1}{2\eta} \left( -||w_t - w^* - \eta v_t||^2 + ||w_t - w^*||^2 + \eta^2 ||v_t||^2 \right)
\]

\[
\omega_{t+1} = w_t - \eta v_t
\]

\[
\beta_{t+1} = \frac{1}{2} \left( -a_{t+1} + a_t \right) + \frac{\eta}{2} ||v_t||^2
\]

\[
\sum_{t=1}^{\infty} (w_t - w^*) \cdot v_t = \frac{1}{2\eta} (\alpha_1 - \alpha_{t+1}) + \frac{\eta}{2} \leq ||v_t||^2
\]
\[ \frac{1}{2\eta} \left( a_t - a_{t+1} \right) + \frac{\eta}{2} \leq \|v_t\|^2 \]

\[ a_{t+1} \|w_{t+1} - w^*\|^2 \]

\[ w_1 = 0 \]

\[ \|w^*\| \leq B \]

\[ \eta = \frac{B}{2\sqrt{T}} \]

\[ \|v_t\| \leq x \]
Lemma 2: If $f$ is convex, differentiable, and $v_t = \nabla f(w_t)$, then

$$\sum_t (f(w_t) - f(w^*)) \leq \sum_t (w_t - w^*) \cdot v_t$$

Proof:

$$\forall x \quad f(w) + \nabla f(w) \cdot (x - w) \leq f(x)$$

$$v_t = \nabla f(w_t) \quad w = w_t \quad x = w^*$$
GD corollary

From Lemma 1: \( \sum_{t=1}^{T} (w_t - w^*) \cdot v_t \leq B\alpha\sqrt{T} \)

From Lemma 2: \( \sum_t (f(w_t) - f(w^*)) \leq \sum_t (w_t - w^*) \cdot v_t \)

The following corollary follows.

GD Guarantee

If \( v_t = \nabla \hat{L}(w_t) \) and \( \hat{L}(w) \) convex and \( \alpha \)-Lipschitz, for every \( w^* \) with \( \|w^*\| \leq B \) (including an optimal \( w^* \)), GD, applied with \( \eta = \frac{B}{\alpha\sqrt{T}} \), satisfies

\[
\hat{L}(\bar{w}) - \hat{L}(w^*) \leq \frac{B\alpha}{\sqrt{T}},
\]

where \( \bar{w} = \frac{1}{T} \sum_t w_t \).
Lemma 1: \[ \sum_{t=1}^{T} (w_t - w^*) \cdot v_t \leq B\alpha\sqrt{T} \]

Lemma 2: \[ \sum_{t} (f(w_t) - f(w^*)) \leq \sum_{t} (w_t - w^*) \cdot v_t \]

\[
\implies \sum_{t} (f(w_t) - f(w^*)) \leq B\alpha\sqrt{T}
\]
SGD analysis

From Lemma 1:

\[ \sum_{t=1}^{T} (w_t - w^*) \cdot v_t \leq B\alpha \sqrt{T} \]

\[ E_{v_1, \ldots, v_T} \left\{ \sum_{t=1}^{T} (w_t - w^*) \cdot v_t \right\} \leq B\alpha \sqrt{T} \]
SGD analysis

\[
\sum_{t=1}^{T} E_{v_1,\ldots,v_t} \{(w_t - w^*) \cdot v_t\} \leq B\alpha\sqrt{T}
\]

**Law of total expectation:** For random variables \(a, b\), and function \(g\):

\[
E_a\{g(a)\} = E_bE_a\{g(a) \mid b\}
\]

\[
E_a(a) = E_bE_a(\cdot) b, a \overset{\text{i.i.d.}}{\sim}
\]
SGD analysis

The inner expectation:

$$(w_t - w^*) \cdot E_{v_t} \{v_t\} = (w_t - w^*) \mathbb{E}_{(x_t, y_t)} \nabla \ell(x_t, y_t, w_t)$$

Therefore,

$$T \mathbb{E}_{v_1, \ldots, v_t} \{(w_t - w^*) \cdot v_t\} = \sum_{t=1}^{T} E_{v_1, \ldots, v_t} \{(w_t - w^*) \cdot v_t\} = \sum_{t=1}^{T} E_{v_1, \ldots, v_{t-1}} \{(w_t - w^*) \cdot \nabla L_D(w_t)\} = E_{v_1, \ldots, v_{T-1}} \sum_{t=1}^{T} (w_t - w^*) \cdot \nabla L_D(w_t) \leq B\alpha\sqrt{T}$$
We thus have,

\[ E_{\mathbf{v}_1, \ldots, \mathbf{v}_{T-1}} \sum_{t=1}^{T} (\mathbf{w}_t - \mathbf{w}^*) \cdot \nabla L_D(\mathbf{w}_t) \leq B\alpha\sqrt{T} \]

From convexity of \( L_D(\mathbf{w}) \)

\[ E_{\mathbf{v}_1, \ldots, \mathbf{v}_{T-1}} \sum_{t=1}^{T} (L_D(\mathbf{w}_t) - L_D(\mathbf{w}^*)) \leq B\alpha\sqrt{T} \]

\[ E \left\{ L_D(\mathbf{\overline{w}}) - L_D(\mathbf{w}^*) \right\} \leq \frac{\alpha}{\sqrt{T}} \]
Averaged-SGD, SGD and GD
Artificial neural networks (ANNs)
Artificial neural networks (ANNs)

- We have seen several types of neuron units: linear, tanh, sigmoid, Relu
- Informally a neural network (ANN) is a directed graph whose nodes are neurons
- A formal (somewhat restricted) definition will be given soon in terms of a computation graph

There are more general definitions of NNs as dynamical systems. For example, we can define a dynamic NN by a weighted directed graph $G = \{V \cup S, E\}$, where $V$ is a set of neurons and $S$ is a set of inputs (e.g., external sensor readings). The dynamics is defined for discrete times $(t)$ and the value of neuron $v \in V$ is defined by

$$v(t + 1) = a \left( \sum_{u \rightarrow v} w_{uv}(t)u(t) + \sum_{s \rightarrow v} w_{sv}(t)s(t) \right)$$
Network types: Feed-forward

- Assuming a NN definition in terms of a directed graph $G$, let’s identify several types of architectures.
- A network defined by a directed acyclic graph (DAG) is called **feed-forward**
- Example:
A feed-forward network whose graph is **layered** is called **multi-layered network**. The set $V$ of neurons is a partition $V = \bigcup_{i=0}^{L} V_i$ and there are edges only between adjacent layers.

In a multi-layered network: the set $V_i$ is called the $i$th layer, layer 0 is called the **input layer**, layer $L$ is the **output layer**, and all other layers are called **hidden**.

For a multi-layered network we can define its **depth** $= L - 1$, **size** $=$ number of non-input neurons), and **width** $= \max_i |V_i|$.

\[
\begin{align*}
S_6 &= 7 = \frac{7}{2} = 3.5 \\
S_{36} &= \left(\frac{36}{2}\right) \cdot \frac{1}{2} = 18 \\
S_{193} &= \left(\frac{193}{2}\right) \cdot \frac{1}{2} = 31.5
\end{align*}
\]
A recurrent neural network (RNN) is defined by a recurring block of neurons.

Models time series
Biological inspiration?

Biological brain is extensively studied but extremely not well understood. Similarities to ANNs are vague and conceptual at best. Consider some trivia-types facts

- Human brain contains 80-100 billion neurons
  - Cockroach: 1,000,000
  - Cat: 760,000,000
  - Grey parrot: 1,566,000,000
  - Dog: 2,253,000,000
  - Chimpanzee: 28,000,000,000
  - African elephant: 257,000,000,000

- Neuron synaptic inputs: up to 10,000. # “weights”: $10^{16}$ (largest ANN to-date: $10^{12}$)

- Early visual cortex neurons receive many non-feedforward inputs from lateral and top-down connections, including long-range projections from auditory areas.
3D imaging of a few neurons
Neural networks as a hypothesis class

- Given a NN definition (in terms of a directed graph $G$), we can define the **NN hypothesis class** as the set of all possible parameters (weights, biases, etc.) that define networks whose graph is $G$. I.e., these networks have the same “architecture”

- Thus, each **architecture defines a hypothesis class**. We will later see that it can encode valuable prior knowledge

- We begin by considering **computation**: how can we train a given architecture?

- For a given architecture we are also interested to determine its

  - **Approximation** error (how expressive it is)
  - **Estimation** error (its sample complexity)
  - **Computation** error (what can be guaranteed by a learning algorithm)

  Note that estimation and computation errors depend on the training algorithm
Computation graph
The computation graph can be used as effective functional description of NNs, and NN training.

Provides a mechanistic way to compute the value of the network for given inputs (forward).

And a mechanistic way to compute gradients of all free parameters (backward).

Why do we need gradients? In our context we will apply SGD (or GD) optimization.

The computation graph is the backbone of the backpropagation algorithm – currently the primary training algorithm for NNs.
Computation graph definition

- The computation graph of a (multivariate, vector) function $f$ is a **directed (acyclic) graph (DAG)** $G = (V, E)$.
- A node $v \in V$ corresponds to an **operation**, a **function**, a **variable**, or a fixed **input**.
- The **output** of a node is dependent on its type: if it is a variable (or a fixed input), the output is the variable itself. Otherwise, the output is the application of the operation or function associated with the node on all its inputs.
- There is a **directed edge** $u \rightarrow v$, if the value of node $u$ is an input of (the operation associated with) node $v$.
- **Terminal nodes** $v$ (whose $d_{out}(v) = 0$) are output values of $f$ (or the components $f$, if it is a vector function).
- The **computation** of the associated function $f$ is conducted using a topological order.
- In general, the input and output of a node is a **tensor**.
Each internal node $v$ in a computation graph should compute:

- The value of its associated function/operation, $v(p_1, \ldots, p_k)$, where $p_i$ are its parents (inputs)
- The partial derivatives $\frac{\partial v}{\partial p_i}$
- The gradient $\frac{\partial L}{\partial v}$ is a summation of the gradients $\frac{\partial L}{\partial v(i)}$, computed by its child $c_i$
- Node $v$ applies the chain rule to compute $\frac{\partial L}{\partial p_i} = \frac{\partial L}{\partial v} \frac{\partial v}{\partial p_i}$, which is passed to its parent $p_i$
Backpropagation algorithm

We are given a computation graph $G = (V, E)$ (a DAG), representing a function $f$.

- **Forward pass:**
  - Traverse $G$ in a topological order:
  - For each encountered node $v \in V$, whose inputs (parents) are $u_1, \ldots, u_k$ compute $v = f_v(u_1, \ldots, u_k)$, and store the result

- **Backward pass:**
  - For each internal node $v$, initialize gradients $\frac{\partial f}{\partial v} = 0$. For terminal nodes, initialize gradients to 1
  - Traverse $G$ in **reverse** topological order:
  - For each encountered node $v$, its gradient $\frac{\partial f}{\partial v}$ is already computed (by induction)
  - Node $v$ implements $f_v(u_1, \ldots, u_k)$ (and its parents are $u_1, \ldots, u_k$)
  - For each parent $u_i$ of $v$ increment $\frac{\partial f}{\partial u_i}$ by $\frac{\partial f}{\partial v} \frac{\partial v}{\partial u_i}$

- **Output:** gradients of all nodes
Backprop algorithm in matrix form

Backprop algorithm is straightforwardly generalized to vector/matrix operations using the generalized chain rule (covered in tutorial). Recall:

- For $F(x) = f(g(x)) = (f \circ g)(x)$, $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- $g : \mathbb{R}^n \rightarrow \mathbb{R}^k$
- $f : \mathbb{R}^k \rightarrow \mathbb{R}^m$
- Then

$$J_{f \circ g}(a) = J_f(g(a)) \cdot J_g(a),$$

where the Jacobian $J_f(g(a)) \in \mathbb{R}^{m \times k}$ and $J_g(a) \in \mathbb{R}^{k \times n}$
Correctness proof for backpropagation by induction is straightforward.

Can be generalized to vector/matrix functions \((f_v \in \mathbb{R}^m\) or \(f_v \in \mathbb{R}^{m \times n}\)). This case the associated gradients are Jacobians. Use the generalized chain rule.

Why is it effective to compute the gradients in reverse order?