Lecture-5

Date: January 22, 2019
Instructor:Navin Goyal **Scribe:Ramya Burra**

Overparameterized setting

Let k and m be the size of hidden layer and size of training data. Overparameterized setting is a situtation where $k > m$.

Question Can we solve ERM in this setting?

Consider a simple case as shown in the picture below. Let (x_i, y_i) denote the training data.

Goal Compute

- $w_1, w_2, ..., w_k \in \mathbb{R}^d$
- $b_1, b_2, ..., b_k \in \mathbb{R}$
- $a_1, a_2, ..., a_k \in \mathbb{R}$

$$f(x) = \sum a_i \text{ReLU}(< w_i, x > + b_i)$$
$$f(x_i) = y_i \forall i \in [m]$$

Algorithm

Consider all ws are in the same direction. Consider a vertical line and sweep it in the choosen direction such that it touches one point at a time as shown in the figure 1. Also assume $\forall i \neq j$ $(x_i, y_i), (x_j, y_j)$ are atleast ϵ apart. Choose w_1, a_1, b_1 such that

$$a_1 \text{ReLU}(< w_1, x_1 > + b_1) = y_1 \text{ and } ||w_1|| = 1$$

Consider

$$< w_1, x_1 > + b_1 = \epsilon$$
$$a_1 = \frac{y_1}{\epsilon}$$
$$a_2 ReLU(< w_2, x_2 > + b_2) = y_2 - a_1 ReLU(< w_1, x_1 > + b_1)$$

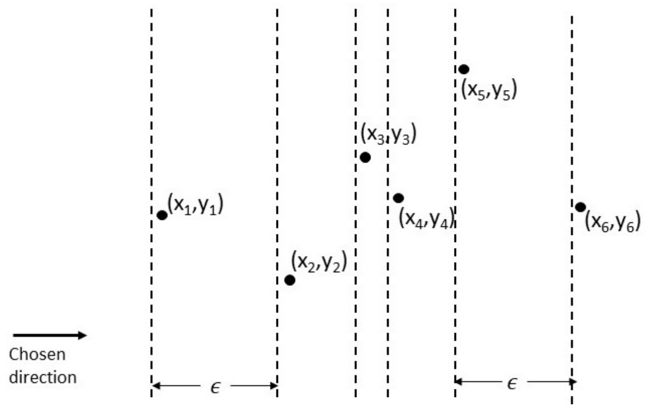

Figure 1: Implementation of algorithm

While choosing w_2, b_2 and a_2, w_1, b_1 and a_1 are left unchanged.

$$f(x_1) = a_1 \text{ReLU}(< w_1, x_1 > +b_1)$$

$$f(x_2) = a_1 \text{ReLU}(< w_1, x_1 > +b_1) + a_2 \text{ReLU}(< w_2, x_2 > +b_2)$$

Remaining w_is can be choosen similarly. Thus ERM can be solved when $k > m$.

Generalization bound

$$L_D(h) \leq L_s(h) + O\left(\sqrt{\frac{VC(\mathbb{H}) + log(\frac{1}{\delta})}{m}}\right)$$

when $k > m$, $VC(\mathbb{H}) > m$ hence

$$L_D(h) \leq L_s(h) + (> 1)$$

This is called vacuous generalization bound.

Minimizing the loss function

To train we perform ERM, that is we try to minimize loss function L for the training examples.

$$\min_{w,b} L_{CE}(w, b, S)$$

Many methods have been invented but perhaps the most popular ones are the simplest ones, they are gradient descent and stochastic gradient descent.

Loss function for Binary class

$$L_{CE}(w, b, (x, y)) = -ylog(\sigma(f_{w,b})(x)) - (1-y)log(1-\sigma(f_{w,b})(x))$$

Loss function for Multi-class

$$L_{CE}(w, b, (x, y)) = -log\left(\frac{e^{f_{w,b}^{(y)}(x)}}{\sum_{y\in[k]} e^{f_{w,b}^{(y)}(x)}}\right)$$

$$L_{CE}(w, b, S) = \sum_{i\in[m]} L_{CE}(w, b, (x_i, y_i))$$

Gradient descent

Algorithm

- Initialize $w^{(0)}$ randomly ($b^{(0)}$ is ignored for simplicity)
- For $i = 1, 2, 3...$ do

$$w^{(i+1)} = w^{(i)} - \eta_i \nabla_w L(w, b, S)$$

- If stopping condition holds then stop otherwise repeat the previous step.

Remark

Gradient descent is also known as full-batch gradient descent. This can have large generalization error, but is easy to implement and easy to parallalize.

Stochastic Gradient descent

Algorithm

- Initialize $w^{(0)}$ randomly ($b^{(0)}$ is ignored for simplicity)
- Sample $S^{(i)} \subset S$ of size s uniformly at random
- For $i = 1, 2, 3...$ do

$$w^{(i+1)} = w^{(i)} - \eta_i \nabla_w L(w, b, S_i)$$

- If stopping condition holds then stop otherwise repeat the previous step.

Remark

Stochastic Gradient descent is more efficient and yields lower generalization error compared to GD.

Efficient and numerically stable computation of the gradient is done by the back propagation algorithm. This is a special case of a more general method called automatic differentiation. Now, we would compute analytical formulas for gradients.
Let us consider binary cross entropy loss function for logestic regression

$$L_{CE}(w, b, (x, y)) = -ylog(\sigma(f_{w,b})(x)) - (1-y)log(1-\sigma(f_{w,b})(x))$$

Properties of σ

$$\frac{d\sigma(t)}{dt} = \sigma(t)(1-\sigma(t))$$

$$\frac{d}{dw_j}log(\sigma(< w, x > +b)) = x_j(1-\sigma)$$

$$\frac{d}{dw_j}log(1-\sigma(< w, x > +b)) = -x_j\sigma$$

$$\frac{d}{dw_j}L = -x_j(\sigma - y)$$

Hence the following are true.

$$\nabla_w L = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_d \end{bmatrix} (\sigma - y) = x(\sigma - y)$$

$$\nabla_w^2 L = xx^T\sigma(1-\sigma) \geq 0$$

Hence this optimization problem is convex.

Back-propagation

Now we would compute derivatives of function compositions. The case of neural networks is a special case. Let us consider a very simple situation of scalar functions.

$$Y = y \circ u \circ v \circ w \circ x$$

where $y, u, v, w : \mathbb{R} \to \mathbb{R}$, $x \in \mathbb{R}$

Goal:

To compute $\frac{dY}{dx}$

- Using the basic definition

$$\frac{Y(x+h)-Y(x)}{h} \approx \frac{dY}{dx}\text{for some } h$$

  This method has issues of truncation and round off error.

- Using chain rule

$$\frac{dY}{dx} = \frac{dY}{du}\frac{du}{dv}\frac{dv}{dw}\frac{dw}{dx}$$

  This method has better numerical properties.

Now, let us consider vector functions

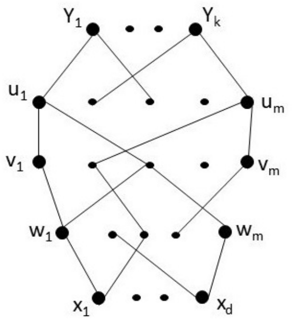

Figure 2: neural network

$$Y_i : \mathbb{R}^m \to \mathbb{R}, i \in [k]$$

$$u_i : \mathbb{R}^m \to \mathbb{R}, i \in [m]$$

$$v_i : \mathbb{R}^m \to \mathbb{R}, i \in [m]$$

$$w_i : \mathbb{R}^m \to \mathbb{R}, i \in [m]$$

$$x_i \in \mathbb{R}, i \in [d]$$

$$Y = y \circ u \circ v \circ w \circ x$$

$$Y_i = Y_i(u_1, u_2, ..., u_m)$$

$$u_j = u_j(w_1, w_2, ..., w_m)$$

Goal: Compute $\frac{\partial Y_j}{\partial x_i}$ for $j \in [k]$ and $i \in [d]$ to compute Jacobian.
Using chain rule we obtain

$$\frac{\partial Y_j}{\partial x_i} = \frac{\partial Y_i}{\partial u_1}(\frac{\partial u_1}{\partial v_1}(\frac{\partial v_1}{\partial w_1}\frac{\partial w_1}{\partial x_i} + \frac{\partial v_1}{\partial w_2}\frac{\partial w_2}{\partial x_i} + ... + \frac{\partial v_1}{\partial w_m}\frac{\partial w_m}{\partial x_i}) + \frac{\partial u_1}{\partial v_2}(...)) + ...$$

$$\frac{\partial Y_j}{x_i} = \sum_{P=(i,i_1,i_2,i_3,j)} \frac{\partial Y_j}{\partial u_{i_3}}\frac{\partial u_{i_3}}{\partial v_{i_2}}\frac{\partial v_{i_2}}{\partial w_{i_1}}\frac{\partial w_{i_1}}{\partial x_j}$$

Explanation: When x_j is changed, the change is reflected in w_is, which in turn changes vs, us and Y_j. The change in x_i can go up to Y_j can go up in all possible ways along the paths P, and some changes are more important if the product of derivatives is large.

Back propagation algorithm

If we calculate product along each path and take the sum, the time will be m^3 and m^L more generally. To do this computationally better we reuse computation. And that is done in back propagation algorithm. Consider

$$\frac{\partial Y_j}{\partial x_i} = \sum_{i_1 \in [m]} \frac{\partial Y_j}{\partial w_{i_1}}\frac{\partial w_{i_1}}{\partial x_j}$$

If we could compute $\frac{\partial Y_j}{\partial w_{i_1}}$ for $i_1 \in [m]$ then we could finish previous computation in time $O(m)$. Now, the next question is how to compute $\frac{\partial Y_j}{\partial w_{i_1}}$?

$$\frac{\partial Y_j}{\partial w_{i_1}} = \sum_{i_2 \in [m]} \frac{\partial Y_j}{\partial v_{i_2}}\frac{\partial v_{i_2}}{\partial w_{i_1}}$$

If we could compute $\frac{\partial Y_j}{\partial v_{i_2}}$ for $i_2 \in [m]$ then we could finish $\frac{\partial Y_j}{\partial w_{i_1}}$ in $O(m^2)$. Now, the next question is how to compute $\frac{\partial Y_j}{\partial v_{i_2}}$?

$$\frac{\partial Y_j}{\partial v_{i_2}} = \sum_{i_3 \in [m]} \frac{\partial Y_j}{\partial u_{i_3}}\frac{\partial u_{i_3}}{\partial v_{i_2}}$$

Now, back propagate. It can be seen that if the total number of edges is E and the total number of nodes is V, then Jacobian can be computed $O(k(V + E))$.