COS-511: Learning Theory **Spring 2017**

Lecture 16: Backpropogation Algorithm

Lecturer: Roi Livni

So far we've discussed convex learning problems. Convex learning problem are of particular interest mainly because they come with strong theoretical guarnatees. For example, we can apply SGD algorithm to obtain desirable learning rates.

As it turns out, even though non-convex problems form formidable challenges in theory: They often tend to solve many interesting problems in practice. In this lecture we will discuss the task of training neural networks using Stochastic Gradient Descent Algorithm. Even though, we cannot guarantee this algorithm will converge to optimum, often state-of-the-art results are obtained by this algorithm and it has become a benchmark algorithm for ML.

16.1 Neural Networks with smooth activation functions

We recall that given a graph (V, E) and an activation function σ we defined $\mathcal{N}_{(V,E),\sigma}$ to be the class of all neural networks implementable by the architecture of (V, E) and activation function σ (See lectures 5 and 6). Given a fixed architecture a target function $f_{\omega,b} \in \mathcal{N}_{(V,E),\sigma}$ is parametrized by a set of weights $\omega : E \to \mathbb{R}$ and bias $b : V \to \mathbb{R}$. The empirical loss (0-1) is given by

$$\mathcal{L}_S^{0,1}(\omega, b) = \sum_{i=1}^{m} \ell_{0,1}(f_{\omega,b}^{(0,1)}(\mathbf{x}^{(i)}), y_i)$$

Where we add the superscript $(0, 1)$ to note that we are considering a target funciton in the class $\mathcal{N}_{(V,E),\sigma_{\sigma_{\text{sgn}}}}$. Of course, the aforementioned problem is non-differentiable (in fact non continous), therefore we cannot apply SGD like method. Therefore we will do two alternations to the architectures considered so far. First instead of $\sigma_{\sigma_{\text{sgn}}}$ that we considered so far we will consider a different activation function. Namely,

$$\sigma(a) = \frac{1}{1 + e^{-a}}.$$

That means that each neuron, now returns as output $\mathbf{v}_i^{(t)} = \sigma(\sum_j \omega_{j,i}^{(t)} v_i^{(t-1)}(\mathbf{x}) + b_j^{(t)})$ which is a smooth function in its parameter. In turn the function $f_{\omega,b}$ becomes smooth in its parameter (since its a composition of addition of smooth functions).

Remark: Note that we care about smoothness in terms of ω and b!!! While $f_{\omega,b}$ is a function of $\mathbf{x}$: In training, we consider the empirical loss as a function of the parameters and we want to optimize over these.

Of course, now the target funciton does not return 0 or 1 but a real number, therefore we also replace the $0 - 1$ with a surrogate convex loss function. For concretness we let $\ell(a, y) = (a - y)^2$. We now obtain the differentiable empirical problem

$$\mathcal{L}_S(\omega, b) = \sum_{i=1}^{m} \ell(f_{\omega,b}(\mathbf{x}^{(i)}), y_i)$$

To see that these alternation do not cause any loss in expressive power or generalization we prove the following claim

Claim 16.1. *Let (V, E) be a fixed feed-forward graph, then for every sample S:*

1. $$\inf \mathcal{L}_S(\omega, b) \leq \inf \mathcal{L}_S^{(0,1)}(\omega, b)$$

2. *For every (ω^*, b^*)*

$$\sum_{i=1}^{m} \ell_{0,1}(\text{sgn}(f_{\omega,b}(\mathbf{x}^{(i)})), y_i) \leq \mathcal{L}_S(\omega^*, b^*)$$

The first claim shows that we can achieve a solution that is competitive with the loss of the optimal neural network with $0-1$ activation function. The second statement tells us that the $0-1$ solution of the optimizer of $\mathcal{L}_S$ will also have small $0-1$ loss. In other words, by minimizing the differentiable problem, we achieve a solution with small empirical $0-1$ loss.

Proof. For the first claim, note that $\lim_{a\to\infty} \sigma(a) = 1$ and $\lim_{a\to-\infty} \sigma(a) = 0$, hence

$$\lim_{c\to\infty} f_{c\cdot\omega, c\cdot b} = f^{0,1}_{\omega,b},$$

hence

$$\lim_{c\to\infty} \mathcal{L}_S(\omega, b) \leq \mathcal{L}_S^{(0,1)}(\omega, b)$$

hence, the first statement hold.

As to the second statement, this follows from the fact that ℓ is a surrogate loss function. □

Thus we turned the non-smooth problem to a differentiable problem. This means that we can now try to apply a gradient descent method, similar to SGD as we used in convex problems. There are two issues to over come

1. Though the loss function might be convex, the ERM problem as a whole, given its dependence on the paratmeter is non convex. We have only shown that SGD converges when the ERM problem is convex in the parameters.
2. To perform SGD we still need to compute the gradient $\nabla f_{\omega,b}$, where the dependence between the parameters may be highly involved.

The first problem turns out to be a real issue and indeed there is no guarantee that SGD will converge to a global optimum when the problem is essentially non-convex. In fact, even convergence to a local minimum is not guaranteed, though one can show that SGD will converge to a critical point (more accurately to a point where $\|\nabla f_{\omega,b}\| \leq \epsilon$ (under certain smoothness assumptions).

The problem is generally solved by re-iterating the algoirhtm from differnet intialization points: with the hope that one of the instances will indeed converge to a sufficiently optimal point. However, all hardness results we discussed so far apply: Therefore for any method, if the network is expressive enough to represent, for example, intersection of halfspaces then for some instances the method must fail.

The second point is actually solvable and we will next see how one can compute the gradient of the loss: This is known as the Backpropagation algorithm, which has become the workhorse of Machine Leanring in the past few years.

16.1.1 A Few Remarks on NNs in practice

Before presenting the Backpropagation algorithm, it's worth discussing some simplifications we have considered here over what is often used in practice:

- **the activation function** We are restricting our attention to a sigmoidal activation function. These has been used in the past. The general intuition being, that they are a smoothing of the $0-1$ activation function. In reality, training with sigmoidal funciton tend to get stuck: when the weights are very large then the derivative starts to behave roughly like the $0-1$ function which mean they vanish. One change that was suggested is to use the relu activation function

  $$\sigma_{\text{relu}} = \max(0, a)$$

  Unlike the sigmoidal funciton, its derivative doesn't vanish whenever the input is positive. In terms of expressive power, they can express sigmoidal like function using

  $$\sigma_{\text{relu}}(a+1) - \sigma_{\text{relu}}(a)$$

  So the overall expressivity of the network doesn't change (as long as we allow twice as many neurons at each layer, which is the same order of neurons)

- **Regularization** For generalization we rely here on the generalization bound of $O(E \log |E|)$. In practice the number of free parameters (weights and bias) tend to be extremely larger then the number of examples. Therefore certain regularization is often employed on the weight (e.g. ℓ_2, ℓ_1 regularization). There have also been other heurisitcs for regulerizing neural networks such as *dropout*: Where roughly, during training one zero out some weights during the update step. As we saw in past lecture SGD comes with its own generalization guarantees. Generalization bounds to SGD for non-convex optimization has been recently obtained in [**?**], but these are not necessarily for the learning rates used in practice.

16.1.2 The Backpropagation Algorithm

We next discuss the Backpropogation algorithm that computes $\frac{\partial f}{\partial \omega, b}$ in linear time.

To simplify and make notations easier, instead of carrying a bias term: let us assume that each layer $V^{(t)}$ contains a single neuron $v_0^{(t)}$ that always outputs a constant 1. thus the output of a neuron is given by $\sigma(\sum \omega_{i,j} v_j^{(t-1)})$ and we supress the bias b as an additional weight $\omega_{i,0}$. We next wish to compute the derivative ∇f_ω. Now suppose neuron $v_i^{(t)}$ computes:

$$v_i^{(t)}(\mathbf{x}) = \sigma(u_i^{(t)}(\mathbf{x}))$$

Where

$$u_i^{(t)}(\mathbf{x}) = \sum \omega_{i,j} v_j^{(t-1)}(\mathbf{x}).$$

Then using a simple chain rule we obtain that

$$\frac{\partial f}{\omega_{i,j}} = \frac{\partial f}{\partial u_i^{(t)}} \cdot \frac{\partial u_i^{(t)}}{\omega_{i,j}} = \frac{\partial f}{\partial u_i^{(t)}} v_j^{(t-1)}(\mathbf{x})$$

Thus to compute the partial derivative with respect to a single weight, we see that it is enough to compute $\frac{\partial f}{\partial u_i^{(t)}}$.

So we focus computing $\frac{f}{\partial u_i^{(t)}}$. Now again suppose f is a function of $u_1^{(t)}, \ldots, u_m^{(t)}$, which are in turn function of some variable z then we have by chain rule:

$$\frac{\partial f}{\partial z} = \sum_{i=1}^{m} \frac{\partial f}{\partial u_i^{(t)}} \cdot \frac{\partial u_i^{(t)}}{\partial z} \tag{16.1}$$

Now if $z = u_n^{(t-1)}$ is the output of some neuron in a previous layer: The calculation of $\frac{\partial u_i^{(t)}}{\partial u_n^{(t-1)}}$ is easy for our choice of activation function

$$u_i^{(t)} = \sum \omega_{i,j} \sigma(u_j^{(t-1)}) \Rightarrow \frac{\partial u_i^{(t)}}{\partial u_n^{(t-1)}} = \omega_{i,n} \sigma'(u_n^{(t-1)}) \quad .$$

Using Eq. 16.1 for $f = u_i^{(t)}$ we can recursively calculate all partial derivative $\frac{\partial u_i^{(t)}}{\partial u_n^{(t')}}$ for $t' < t$, which in turn will give us also $\frac{\partial f_\omega}{\partial u_i^{(t)}}$.

The naive approach to calculate the gradient is that we calculate inductively all derivatives of the form $\frac{\partial u_i^{(t)}}{\partial u_j^{(t')}}$ for $t' < t$, then using Eq. 16.1 with $f = u_i^{(t+1)}$ we calculate all derivatives $\frac{\partial u_\ell^{(t+1)}}{\partial u_j^{(t')}}$. This calculation calls for each fixed j number of times proportional to $|E|$ number of edges, therefore the overall calculation time is given by $O(|V||E|)$. The back propogation algorihtm calculates the derivative through dynamical programming and reduces the complexity to $O(|V| + |E|)$:

16.1.3 Backproporgation

We next consider an approach to calculate the partial derivative that takes time $O(|V| + |E|)$.

Algorithm 1 Backpropogation

Input: Graph $G(V, E)$ and parameters $\omega : E \to \mathbb{R}$.
SET $T = \text{depth}G$, i.e $v^{(T)}$ is the output neuron.
SET $m^{(T)} = 1$
for $t = T - 1 \ldots 1$ % Start from top layer and move toward bottom layer **do**
for $i = 1, \ldots, |V^{(t)}|$ % Go over all neurons at layer t **do**
neuron $v_i^{(t)}$ receive messages $m_j^{(t+1)}(v_i^{(t)})$, sums them up:

$$S = \sum_{j=1}^{V^{(t+1)}} m_j^{(t+1)}(v_i^{(t)})$$

and passes a message $m_i^{(t)}(v_k^{(t-1)})$ to each neuron at a lower level:

$$m_i^{(t)}(v_k^{(t-1)}) = S \cdot \frac{\partial u_i^{(t)}}{\partial u_k^{(t-1)}}$$

end for
end for

Claim 16.2. *At each node $v_i^{(t)}$ the value S is exactly $\frac{\partial f_\omega}{\partial v_i^{(t)}}$.*

Proof. We prove the statement by induction. The message that receives the output neuron is given by $S = \frac{\partial f_\omega}{\partial f_\omega} = 1$. Next for each neuron we have by induction:

$$S = \sum_{j=1}^{|V^{(t+1)}|} \frac{\partial f_\omega}{\partial u_j^{(t+1)}} \cdot \frac{\partial u_i^{(t+1)}}{\partial u_i^{(t)}} \tag{16.2}$$

Which by chain rule gives the desired result □

The Backpropagation, each neuron does number of computation that is proportional to its degree, overall the number of calculation is proportional to twice the number of edges which gives overall number of calculations $O(|V| + |E|)$.