Backpropagation and SGD

Saeed Saremi

Assigned reading: 5.4.4, 8

September 26, 2024

a summary of the previous lecture(s) I

- L-layer (feedforward) neural network as a particular way of composing functions:

$$a_j^{(\ell)} = \sum_{i \in [d_{\ell-1}]} \theta_{ji}^{(\ell)} x_i^{(\ell-1)}$$
$$x_j^{(\ell)} = g_\ell(a_j^{(\ell)}),$$
$$g_\ell : \mathbb{R} \to \mathbb{R},$$

  where the computation is iterative from $\ell = 1$ to $\ell = L$.
- $x^{(0)}$ represents the INPUT, $x^{(L)}$ the OUTPUT: $f_\theta : x^{(0)} \mapsto x^{(L)}$
- by convention $g_L(z) = z$, and $g_\ell = g$ is typically fixed for $\ell \in [L-1]$.
- examples of g include: $z \mapsto 1/(1+e^{-z})$ and $z \mapsto \max(z, 0)$.
- one can increase the "capacity" of the neural network by either increasing L or increasing the number of hidden neurons (and thus connections) in each layer.

a summary of the previous lecture(s) II

- The loss $\mathcal{L}(\theta)$ we minimize is formulated as negative log likelihood.
  - regression, $\mathbb{R}^d \to \mathbb{R}$:

$$\mathcal{L} = \frac{1}{2}(y - a_1^{(L)})^2$$

  - classification, $\mathbb{R}^d \to \{0, 1\}$:

$$\mathcal{L} = -y \log \sigma(a_1^{(L)}) - (1 - y) \log(1 - \sigma(a_1^{(L)})),$$

  where σ is the logistic sigmoid function:

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

- for a 1-layer ($L = 1$) neural network:

$$a_k^{(L)}(x^{(0)}; \theta) = \sum_{i \in [d_0]} \theta_{ki}^{(1)} x_i^{(0)},$$

  where in the two examples above $k = 1$ (only one output neuron).

a summary of the previous lecture(s) III

- In gradient-based optimization we minimize the loss $\mathcal{L}(\theta)$ using gradient descent:

$$\theta_{t+1} = \theta_t - \epsilon_t \nabla_{\theta_t} \mathcal{L}(\theta_t),$$

  where ϵ_t is some "small" number where we can take to be constant $\epsilon_t = \epsilon$.
- For 1-layer ($L = 1$) neural networks $\nabla \mathcal{L}$ takes the following general form:

$$\partial_{\theta^{(1)}_{ki}} \mathcal{L} = \partial_{a^{(L)}_k} \mathcal{L} \; \partial_{\theta^{(1)}_{ki}} a^{(L)}_k$$

  - (gaussian) regression:

$$\partial_{a^{(L)}_1} \mathcal{L} = a^{(L)}_1 - y$$

  - logistic regression:

$$\partial_{a^{(L)}_1} \mathcal{L} = \sigma(a^{L}_1) - y$$

outline I

- $\partial_{a^{(L)}}\mathcal{L}$ has a clear interpretation in terms of the error signal that drives learning.
- We therefore name it as the *error* (signal) and use the following notation:

$$\Delta_k^{(L)} = \partial_{a_k^{(L)}}\mathcal{L}.$$

Next we derive $\Delta_k^{(L)}$ for multi-class ($K > 2$) classification, where the loss is given by

$$\mathcal{L} = -\sum_{k\in[K]} y_k \log \frac{\exp(a_k^{(L)})}{Z},$$

where Z is the normalization:

$$Z = \sum_{k\in[K]} \exp(a_k^{(L)})$$

Can you guess the answer?

"error" in multiclass classification

$$\Delta_k = \frac{\partial \mathcal{L}}{\partial a_k^{(L)}} = -\frac{\partial}{\partial a_k}\left(\sum_{j=1}^{K} y_j \log \underbrace{\frac{e^{a_j}}{Z}}_{p_j}\right)$$

$$Z = \sum_{i=1}^{K} e^{a_i}$$

$$\frac{\partial Z}{\partial a_k} = e^{a_k}$$

$$\frac{\partial p_j}{\partial a_k} = \begin{cases} \dfrac{e^{a_j}}{Z} - \dfrac{e^{a_j} e^{a_j}}{Z^2} = p_j(1-p_j) & j = k \\[2ex] -\dfrac{e^{a_j} e^{a_k}}{Z^2} = -p_j p_k & j \neq k \end{cases}$$

$$= p_j(\delta_{jk} - p_k) \qquad \delta_{jk} = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases}$$

$$\Delta_k = -\sum_{j=1}^{K} y_j \frac{1}{p_j} \frac{\partial p_j}{\partial a_k}$$

$$= -\sum_{j=1}^{K} y_j \frac{1}{\cancel{p_j}} \cancel{p_j}(\delta_{jk} - p_k)$$

$$= -y_k + \underbrace{\left(\sum_{j=1}^{K} y_j\right)}_{1} p_k = p_k - y_k$$

$$(0 \text{ —— } 0 \; 1)$$

outline II

- Derive an algorithm for computing

$$\partial_{\theta_{ji}^{(l)}} \mathcal{L}$$

for all the elements of $\theta = (\theta^{(1)}, \ldots, \theta^{(L)})$ that is of $O(m)$[1] where $m = \dim(\theta)$.

- The error signals

$$\Delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial a_j^{(l)}}$$

plays a central role in the final algorithm!

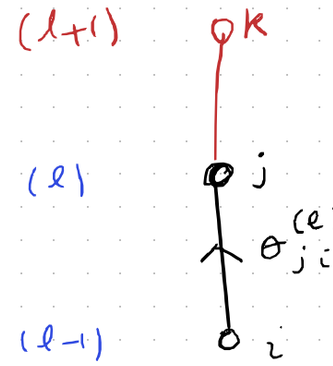
- The algorithm called backpropagation involves computing the errors backward from top $\ell = L$ back to $\ell = 1$ by a single pass through the neural network.[2]

[1] We already know a (doubly bad) algorithm that is of $O(m^2)$:

$$\partial_{\theta_{ji}^{(l)}} \mathcal{L} \approx \frac{\mathcal{L}(\theta_{ji}^{(l)} + \epsilon) - \mathcal{L}(\theta_{ji}^{(l)} - \epsilon)}{2\epsilon}$$

[2] I recommend reading the original paper: D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*, Nature, 1986.

backpropagation



$$\frac{\partial \alpha}{\partial \theta^{(l)}_{ji}}$$

$$a^{(l)}_j = \sum_{i \in [d_{l-1}]} \theta^{(l)}_{ji} \, x^{(l-1)}_i \qquad \begin{array}{l} i \in [d_{l-1}] \\ j \in [d_l] \end{array}$$

$$x^{(l-1)}_i = g\left(a^{(l-1)}_i\right)$$

$$\alpha = \begin{cases} \frac{1}{2}\left(y - a^{(L)}\right)^2 \\ -y \log \sigma\left(a^{(L)}\right) - (1-y) \log\left(1 - \sigma^{(L)}\right) \\ -\sum_{k \in [K]} y_k \log \dfrac{a^{(L)}_k}{\sum_{j \in [K]} e^{a^{(L)}_j}} \end{cases}$$

$$\frac{\partial \alpha}{\partial \theta^{(l)}_{ji}} = \frac{\partial \alpha}{a^{(l)}_j} \frac{\partial a^{(l)}_j}{\partial \theta^{(l)}_{ji}} = \Delta^{(l)}_j \, x^{(l-1)}_i$$

$$\Delta^{(l)}_j := \frac{\partial \alpha}{\partial a^{(l)}_j} = \sum_{k \in [d_{l+1}]} \underbrace{\frac{\partial \alpha}{\partial a^{(l+1)}_k}}_{\Delta^{(l+1)}_k} \frac{\partial a^{(l+1)}_k}{\partial a^{(l)}_j}$$

$$\Delta^{(l)}_j = \sum_{k \in [d_{l+1}]} \Delta^{(l+1)}_k \frac{\partial a^{(l+1)}_k}{\partial a^{(l)}_j}$$

$$a^{(l+1)}_k = \sum_{j \in [d_l]} \theta^{(l+1)}_{kj} \underbrace{g\left(a^{(l)}_j\right)}_{x^{(l)}_i}$$

$$\frac{\partial a^{(l+1)}_k}{\partial a^{(l)}_j} = \theta^{(l+1)}_{kj} \, g'\left(a^{(l)}_j\right)$$

$$\boxed{\Delta^{(l)}_j = \left(\sum_{k \in [d_{l+1}]} \Delta^{(l+1)}_k \, \theta^{(l+1)}_{kj}\right) g'\left(a^{(l)}_j\right)}$$

stochastic gradient descent I

- at a high level the loss is always written as the sum of losses by individual points $i \in [n] := \{1, \ldots, n\}$ in the training set $\mathcal{D} = \{(x_i, y_i)\}_{i \in [n]}$:

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \mathcal{L}_i(\theta),$$

  where $\mathcal{L}_i(\theta)$ is short for:

$$\mathcal{L}_i(\theta) := \mathcal{L}(x_i, y_i; \theta).$$

- This is very general, but it's very easy to see where it's coming from in the maximum log-likelihood framework. We always assume the i.i.d. setting:

$$(x_i, y_i) \overset{\text{iid}}{\sim} p_\theta(x, y), \ i \in [n].$$

  Therefore,

$$p(\mathcal{D}|\theta) = \prod_{i=1}^{n} p_\theta(x_i, y_i).$$

  It follows:

$$\mathcal{L}_i(\theta) = -\log p_\theta(x_i, y_i).$$

stochastic gradient descent II

Stochastic Gradient Descent (SGD) in its pure form is defined by the following updates:

$$\theta_{t+1} = \theta_t - \epsilon_t \nabla_{\theta_t} \mathcal{L}_i(\theta_t),$$

where $i \in [n]$ is selected at random (typically without replacement) at each iteration t.

stochastic gradient descent III

- Intuitively (people have tried to study this) in high dimensions the loss landscape is "dominated" by saddle points and the noise in SGD helps to avoid them.
- SGD is convenient in the regime $n \ggg 1$.
- One pass through the data is called an epoch.
- The problem is towards the end of the training (optimization) the noise in SGD will slow down the training.
- In short: noise helps us at the beginning of training, it "hurts" us towards the end.
- Of course, one can find a compromise by dividing the dataset into (random) mini-batches of size b: in this scheme one epoch involves $\lfloor n/b \rfloor$ updates.

? Based on this picture, can you suggest a batching scheme for effective training?[3]

[3] Coming up with a mini-batch/learning rate schedule remains an art and it is problem dependent.