CSCI 5525 Machine Learning Fall 2019

Lecture 10: Optimization and Training

Oct 3rd 2019

Lecturer: Steven Wu *Scribe: Steven Wu*

1 Backpropagation

In this lecture, we will talk about how to learn the parameters of a neural network. Recall that a neural network maps an input vector x to

$$F(x, \theta) = \sigma_L \left(W_L(\dots W_2 \sigma_1(W_1 x + b_1) + b_2 \dots) + b_L \right)$$

where θ contains all of the parameters $W_1, \dots, W_L, b_1, \dots, b_L$. The relevant ERM problem is then defined as minimizing the following empirical risk function over θ:

$$\hat{\mathcal{R}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, F(x_i, \theta))$$

We will use gradient descent method to optimize this function, even though the loss function is non-convex. First, the graident w.r.t. each W_j is defined as

$$\nabla_{W_j} \hat{\mathcal{R}}(\theta) = \nabla_{W_j} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, F(x_i, \theta)) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{W_j} \ell(y_i, F(x_i, \theta))$$

We can derive the same equality for the gradient w.r.t. each b_j. It suffices to look at the gradient for each example. We can rewrite the loss for each example as

$$\begin{aligned} \ell(y_i, F(x_i, \theta)) &= \ell(y_i, \sigma_L \left(W_L(\dots W_2 \sigma_1(W_1 x + b_1) + b_2 \dots) + b_L \right)) \\ &= \tilde{\sigma}_L \left(W_L(\dots W_2 \sigma_1(W_1 x + b_1) + b_2 \dots) + b_L \right) \end{aligned}$$

where $\tilde{\sigma}_L$ absorbs y_i and ℓ, that is $\tilde{\sigma}_L(a) = \ell(y_i, a)$ for any a. Note that σ_L' can just be viewed as another activation function, so this loss function can just be viewed as a different neural network mapping. It suffices to look at the gradient $\nabla_{W_j} F(x, \theta)$, whose gradient will have the same form as the gradient of the loss.

Backpropagation is a linear time algorithm with runtime $O(V + E)$, where V is the number of nodes and E is the number of edges in the network. The idea is a message passing protocol: each node in the network u receives a message from the neighbor v along each outgoing edge such that each message. Then the node u sums these messages to get a number S, and further delivers the following message to any node z adjacent to it at a lower level: $S \cdot \frac{\partial u}{\partial z}$.

Let's work out the case where everything is in $\mathbb{R}$.

$$F_j(\theta) = \sigma_j\left(W_j(\dots W_2\sigma_1(W_1x + b_1) + b_2 \dots) + b_j\right) \qquad J_j = \sigma_j'(W_j\sigma_{j-1}(\theta) + b_j)$$

Let's apply chain rule from top to bottom.

$$\begin{aligned}
\frac{\partial F_L}{\partial W_L} &= J_L F_{L-1}(\theta) \\
\frac{\partial F_L}{\partial b_L} &= J_L \\
&\dots \\
\frac{\partial F_L}{\partial W_j} &= J_L W_L J_{L-1} W_{L-1} \dots F_{j-1}(\theta) \\
\frac{\partial F_L}{\partial b_j} &= J_L W_L J_{L-1} W_{L-1} \dots J_j
\end{aligned}$$

That looks nice and simple. Now as we move to multi-dimensional case, we will need the following multivariate chain rule:

$$\nabla_W f(Wa) = J^\intercal a^\intercal$$

where $J \in \mathbb{R}^l \times \mathbb{R}^k$ is the Jacobian matrix of $f\colon \mathbb{R}^k \to \mathbb{R}^l$ at Wa. This gives

$$\begin{aligned}
\frac{\partial F_L}{\partial W_L} &= J_L^\intercal F_{L-1}(\theta)^\intercal \\
\frac{\partial F_L}{\partial b_L} &= J_L^\intercal \\
&\dots \\
\frac{\partial F_L}{\partial W_j} &= (J_L W_L J_{L-1} W_{L-1} \dots J_j)^\intercal F_{j-1}(\theta)^\intercal \\
\frac{\partial F_L}{\partial b_j} &= (J_L W_L J_{L-1} W_{L-1} \dots J_j)^\intercal
\end{aligned}$$

where J_j is the Jacobian of σ_j at $W_jF_{j-1}(\theta) + b_j$. If σ_j is applying the coordiantewise activation function, then the Jacobian matrix is diagonal.

2 Stochastic Gradient Descent

Recall that the empirical gradient is defined as

$$\nabla_\theta \hat{\mathcal{R}}(\theta) = \nabla_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, F(x_i, \theta))$$

For large n, this can be very expensive to compute. A common practice is to evaluate the gradient on a *mini-batch* $\{(x_i', y_i')\}_{i=1}^b$ selected uniformly at random. In expectation, the update is moving to the right direction:

$$\mathbf{E}\left[\frac{1}{b}\sum_i \nabla_\theta \ell(y_i', F(x_i, \theta^t))\right] = \nabla_\theta \hat{\mathcal{R}}(\theta^t)$$

The batch size is another hyperparameter to tune.

3 Alternatives to gradient descent

Nesterov Acceleration Initially: let $v_0 = 0$ and w_0 be arbitrary. The update step:

$$v_{i+1} = w_i - \eta \nabla_w F(w)$$
$$w_{i+1} = v_{i+1} + \frac{i+1}{i+4}(v_{i+1} - v_i)$$

Newton methods Newton iteration:

$$w' = w - \eta \left(\nabla^2 F(w)\right)^{-1} \nabla F(w)$$

An advantage of this method is that it does not have a learning rate η, which saves some tuning. For certain cases, this converges faster in terms of number of iterations. However, per-iteration computation involves computing the Hessian matrix, which can be prohibitively expensive.

4 Training tricks

Random initialization. Initializing all the weights to be zero is a bad idea in general, since all the neurons will be computing the same functions even after gradient descent udpates. Common practice is to randomly initialize the weights.

Batch normalization A useful technique that seems to accelerate training of neural networks is *batch normalization*, which performs a standardization of the node outputs. Standardization is a method of rescaling the feature. For each feature j and a batch of data $x_1, x_2, \ldots x_m \in \mathbb{R}^d$, we can transform the feature as

$$\tilde{x}_{ij} = \frac{x_{ij} - \overline{x}_j}{\sigma_j}$$

where $\overline{x}_j = \frac{1}{m}\sum_i x_{ij}$ and $\sigma_j^2 = \frac{1}{m}\sum_i x_{ij}^2$. The rescaled feature vectors will then have mean zero, and unit variance in each coordinate. This is a useful technique for training linear models for linear regression or logistic regression. Sometimes we replace σ_j in the denominator by $\sqrt{\sigma_j^2 + \epsilon}$ for some small value of ϵ. Batch normalization is simply applying the standardization method to the input to the activation function at each node.

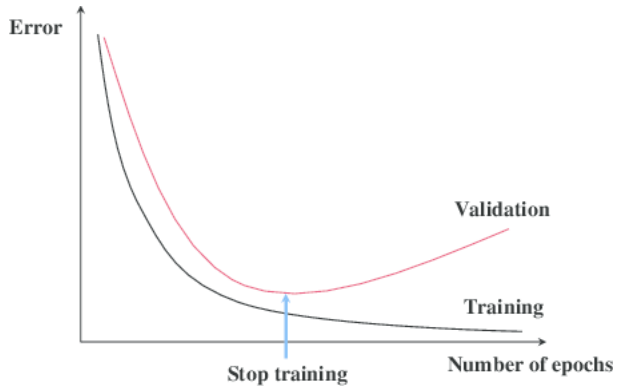

Figure 1: Early stopping. Image source.

Early stopping. Stop training when the performance on validation set stops improving. This is a useful way to mitigate overfitting.