Handout 16: Training Neural Networks

Gradient Descent

Gradient descent is an iterative algorithm for finding the minimum value of a function $f(b)$. It updates to a new value by the formula

$$b_{new} = b_{old} - \eta \cdot \nabla_b f(b_{old}), \tag{11.1}$$

for a fixed learning rate η. At each step it moves in the direction the function locally appears to be decreasing the fastest, at a rate proportional to how fast it seems to be decreasing. Gradient descent is a good algorithm choice for neural networks. Faster second-order methods involve the Hessian matrix, which requires the computation of a square matrix with dimensions equal to the number of unknown parameters. Even relatively small neural networks have tens of thousands of parameters making storage and computation of the Hessian matrix infeasible. Conversely, we will see today that the gradient can be computed relatively quickly.

Stochastic Gradient Descent (SGD)

Neural networks generally need many thousands of iterations to converge to a reasonable minimizer of the loss function. With large datasets and models, while still feasible, gradient descent can become quite slow. Stochastic gradient descent (SGD) is a way of incrementally updating the weights in the model without needing to work with the entire dataset at each step. To understand the derivation of SGD, first consider updates in ordinary gradient descent:

$$\left(b^{(0)} - \eta \cdot \nabla_b f\right) \;\rightarrow\; b^{(1)} \tag{11.2}$$

Notice that for squared error loss (it is also true for most other loss functions), the loss can be written as a sum of component losses for each observation. The gradient, therefore, can also be written as a sum of terms over all of the data points.

$$\begin{aligned} f(b) &= \sum_i (\widehat{y}_i(b) - y_i)^2 && (11.3)\\ &= \sum_i f_i(b) && (11.4)\\ \nabla_w f &= \sum_i \nabla_w f_i && (11.5)\end{aligned}$$

This means that we could write gradient descent as a series of n steps over each of the training observations.

$$\begin{aligned} \left(b^{(0)} - (\eta/n) \cdot \nabla_{b^{(0)}} f_1\right) &\;\rightarrow\; b^{(1)} && (11.6)\\ \left(b^{(1)} - (\eta/n) \cdot \nabla_{b^{(0)}} f_2\right) &\;\rightarrow\; b^{(2)} && (11.7)\\ &\vdots && (11.8)\\ \left(b^{(n-1)} - (\eta/n) \cdot \nabla_{b^{(0)}} f_n\right) &\;\rightarrow\; b^{(n)} && (11.9)\\ & && (11.10)\end{aligned}$$

The output $b^{(n)}$ here is exactly equivalent to the $b^{(1)}$ from before.

The SGD algorithm actually does the updates in an iterative fashion, but makes one small modification. In each step it updates the gradient with respect to the new set of weights. Writing η' as η divided by the sample size, we can write this as:

$$\left(b^{(0)} - \eta' \cdot \nabla_{b^{(0)}} f_1\right) \rightarrow b^{(1)} \tag{11.11}$$

$$\left(b^{(1)} - \eta' \cdot \nabla_{b^{(1)}} f_2\right) \rightarrow b^{(2)} \tag{11.12}$$

$$\vdots \tag{11.13}$$

$$\left(b^{(n-1)} - \eta' \cdot \nabla_{b^{(n)}} f_n\right) \rightarrow b^{(n)} \tag{11.14}$$

In comparison to the standard gradient descent algorithm, the approach of SGD should seem reasonable. Why work with old weights in each step when we already know what direction the vector b is moving? Notice that SGD does not involve any stochastic features other than being sensitive to the ordering of the dataset. The name is an anachronism stemming from the original paper of Robbins and Monro which suggested randomly selecting the data point in each step instead of cycling through all of the training data in one go.

In the language of neural networks, one pass through the entire dataset is called an *epoch*. It results in as many iterations as there are observations in the training set. A common variant of SGD, and the most frequently used in the training of neural networks, modifies the procedure to something between pure gradient descent and pure SGD. Training data are grouped into mini-batches, typically of about 32–64 points, with gradients computed and parameters updated over the entire mini-batch. The benefits of this tweak are two-fold. First, it allows for faster computations as we can vectorize the gradient calculations of the entire mini-batch. Secondly, there is also empirical research suggesting that the mini-batch approach stops the SGD algorithm from getting stuck in saddle points.

Backwards Propagation of Errors

In order to apply SGD to neural networks, we need to be able to compute the gradient of the loss function with respect to all of the trainable parameters in the model. For dense neural networks, the relationship between any parameter and the loss is given by the composition of linear functions, the activation function σ, and the chosen loss function. Given that activation and loss functions are generally well-behaved, in theory computing the gradient function should be straightforward for a given network. However, recall that we need to have thousands of iterations in the SGD algorithm and that even small neural networks typically have thousands of parameters. An algorithm for computing gradients as efficiently as possible is essential. We also want an algorithm that can be coded in a generic way that can then be used for models with an arbitrary number of layers and neurons in each layer.

The backwards propagation of errors, or just *backpropagation*, is the standard algorithm for computing gradients in a neural network. It is conceptually based on applying the chain rule to each layer of the network in reverse order. The first step consists in inserting an input x into the first layer and then propagating the outputs of each hidden layer through to the final output. All of the intermediate outputs are stored. Derivatives with respect to parameters in the last layer are calculated directly. Then, derivatives are calculated showing how changing the parameters in any internal layer affect the output of just that layer. The chain rule is then used to compute the full gradient in terms of these intermediate quantities with one pass backwards through the network. The conceptual idea behind backpropagation can be applied to any hierarchical model described by a directed acyclic graph (DAG). You will now work out the algorithm for a simple example with just a single set of hidden neurons.

Exercises

Assume that we have a neural network with one hidden layer, defined as (where x is one row of the input matrix, y is the corresponding output value, and we have a layer of T hidden nodes in the network):

$$z_k = \alpha_k + \sum_{j=1}^{P} B_{j,k} \cdot x_j, \quad k = 1, \ldots, T \tag{11.15}$$

$$a_k = \sigma(z_k), \quad k = 1, \ldots, T \tag{11.16}$$

$$w = c + \sum_{k=1}^{T} \gamma_k a_k \tag{11.17}$$

Where σ is a differentiable activation function. The terms c, γ_k, and $B_{j,k}$ are the parameters that define the model. We want to minimize the quantity (the loss function):

$$L(w, y) = \frac{1}{2} \cdot (w - y)^2. \tag{11.18}$$

We need to compute a number of partial derivatives, which we will do using the chain rule. It is important that you don't jump ahead and plug things in before I ask you to.

Step 1: Compute the partial derivative of:

$$\frac{\partial z_k}{\partial B_{j,k}} =$$

Note that z_k does not depend on $B_{j,m}$ if $m \neq k$, so we do not need to worry about those terms.

Step 2: Compute the partial derivative of (yes, this is easy):

$$\frac{\partial z_k}{\partial \alpha_k} =$$

Step 3: Write down a formula for the following using the notation $\sigma'(\cdot)$ to denote the derivative of σ.

$$\frac{\partial a_k}{\partial z_k} =$$

Step 4: Write down a formula for:

$$\frac{\partial w}{\partial \gamma_k} =$$

Step 5: What is the following (yes, this is easy also):

$$\frac{\partial w}{\partial c} =$$

Step 6: Finally, what is the derivative of the loss function with respect to w:

$$\frac{\partial L}{\partial w} =$$

Step 7: Notice that I can use the chain rule to write the following, the derivative with respect to each tunable parameter in the second layer of the model:

$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial w} \cdot \frac{\partial w}{\partial c} \tag{11.19}$$
$$\frac{\partial L}{\partial \gamma_k} = \frac{\partial L}{\partial w} \cdot \frac{\partial w}{\partial \gamma_k} \tag{11.20}$$

Now, plug in the values that you know to compute each of these:

$$\frac{\partial L}{\partial c} =$$
$$\frac{\partial L}{\partial \gamma_k} =$$

Step 8: What about the terms $B_{j,k}$ and α_k? They are in the hidden layer and require one more step:

$$\frac{\partial L}{\partial B_{j,k}} = \frac{\partial L}{\partial z_k} \cdot \frac{\partial z_k}{\partial B_{j,k}} \tag{11.21}$$
$$= \frac{\partial L}{\partial w} \cdot \frac{\partial w}{\partial a_k} \cdot \frac{\partial a_k}{\partial z_k} \cdot \frac{\partial z_k}{\partial B_{j,k}} \tag{11.22}$$

And:

$$\frac{\partial L}{\partial \alpha_k} = \frac{\partial L}{\partial z_k} \cdot \frac{\partial z_k}{\partial B_{j,k}} \tag{11.23}$$
$$= \frac{\partial L}{\partial w} \cdot \frac{\partial w}{\partial a_k} \cdot \frac{\partial a_k}{\partial z_k} \cdot \frac{\partial z_k}{\partial \alpha_k} \tag{11.24}$$

But, you do know all of these terms. Plug them in to get the partial derivative with respect to $B_{j,k}$ and α_k.

$$\frac{\partial L}{\partial B_{j,k}} =$$
$$\frac{\partial L}{\partial \alpha_k} =$$

Summary

The most important lines to understand here are Equations 11.21 and 11.23. They show the core back propagation logic: decomposing the influence of a parameter to (i) how it influences the output of that layer and (ii) how that layer influences the loss. This make it possible, with just a little bit more notation, to compute gradients for the deepest of neural networks.