Appendix A

Back-propagation Algorithm

Our grammar correction system is based on a data-driven deep learning approach. DGC learns to abstract features corresponding to the grammar of the language by means of gradient descent optimization within neural networks. The algorithm used to achieve this is called back-propagation. We will review the basics of back-propagation learning below.

Suppose we have a neural network $\mathcal{N}$, which computes some function $f : X \to Y$. Given a set (called training set) $\mathbb{S} \subset X \times Y$ of correct input-output pairs, we would like to adjust the weights of $\mathcal{N}$ in order to better match the training set. In order to define the meaning of "better" we first introduce a measure of network's performance, called a loss function (sometimes the term error function is also used) $L^f : \mathcal{P}(X \times Y) \to \mathbb{R}$. In order for back-propagation to work, two basic assumptions about L have to be noted. First, it should decompose as $L^f(\mathbb{S}) = \frac{1}{n} \sum_x L^f_x$, where each L^f_x represents loss on a single training example $(x, y) \in \mathbb{S}$, and $n = |S|$. Second, it should be a function of the outputs from $\mathcal{N}$. The loss function is used to compute how far the network's predictions, for inputs in the training set, differ from the correct answers. A standard choice is quadratic loss

$$L^f(\mathbb{S}) = \frac{1}{2n} \sum_x (y - \hat{y})^2$$

where $\hat{y}$ denotes the network's prediction (i.e. $f(x)$) for the input example (x, y). Other types of loss can be motivated by statistical or information theoretical considerations (e.g. we could compute KL-divergence between the distribution of the network's outputs and the training examples). The back-propagation algorithm optimizes loss on the training set via stochastic gradient descent.

Here, we will derive the back-propagation update rules using a simple neural network model shown in figure A.1. In the diagram of figure A.1, squares denote input neurons (hence no activation function is applied there), and circles denote sigmoid neurons. Neurons are arranged in layers, which are denoted by named boxes I, J, K. We will use the following notation

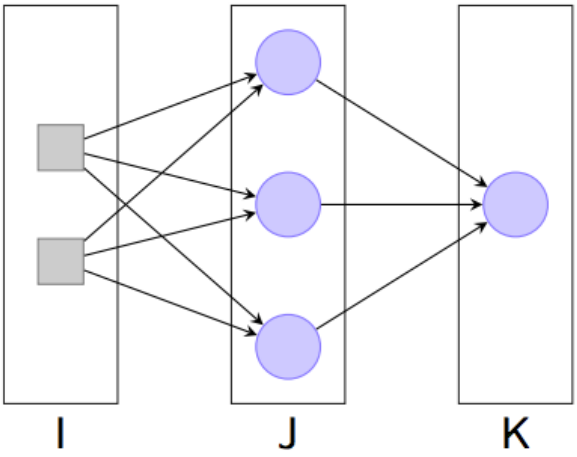

Figure A.1: A simple neural network

- x_j^l : input to node j of layer l
- W_{ij}^l : weight from layer $l-1$ node i to layer l node j
- $\sigma(x) = \frac{1}{1+e^{-x}}$: sigmoid transfer function
- θ_j^l : bias of node j in layer l
- $\mathcal{O}_j^l$: output of node j in layer l
- τ_j : target value of node j in layer l
- $E = \frac{1}{2} \sum_{k \in K} (\mathcal{O}_k - \tau_k)^2$: loss

In order to derive the back-propagation algorithm, we need to compute the derivative of our loss with respect to weights and biases of each node in the network. We split our computation into two cases.

First let us consider the output layer (K).

$$\begin{aligned}
\frac{\partial E}{\partial W_{jk}} &= \frac{\partial}{\partial W_{jk}} \frac{1}{2} \sum_{k' \in K} (\mathcal{O}_{k'} - \tau_{k'})^2 \\
&= (\mathcal{O}_k - \tau_k) \frac{\partial}{\partial W_{jk}} \mathcal{O}_k \\
&= (\mathcal{O}_k - \tau_k) \sigma(x_k)(1 - \sigma(x_k)) \frac{\partial}{\partial W_{jk}} x_k \\
&= (\mathcal{O}_k - \tau_k) \mathcal{O}_k (1 - \mathcal{O}_k) \mathcal{O}_j \\
&= \mathcal{O}_j \delta_k
\end{aligned}$$

where we defined $\delta_k = (\mathcal{O}_k - \tau_k)\mathcal{O}_k(1 - \mathcal{O}_k)$ which is basically the difference between the example and the prediction, scaled by the derivative of the transfer function.

Similarly, we compute the gradient of the loss function for the hidden layer (J).

$$\begin{aligned}
\frac{\partial E}{\partial W_{ij}} &= \frac{\partial}{\partial W_{ij}} \frac{1}{2} \sum_{k \in K} (\mathcal{O}_k - \tau_k)^2 \\
&= \sum_{k \in K} (\mathcal{O}_k - \tau_k) \frac{\partial}{\partial W_{ij}} \mathcal{O}_k \\
&= \sum_{k \in K} (\mathcal{O}_k - \tau_k) \sigma(x_k)(1 - \sigma(x_k)) \frac{\partial}{\partial W_{ij}} x_k \\
&= \sum_{k \in K} (\mathcal{O}_k - \tau_k) \mathcal{O}_k (1 - \mathcal{O}_k) \frac{\partial x_k}{\partial \mathcal{O}_j} \frac{\partial \mathcal{O}_j}{\partial W_{ij}} \\
&= \frac{\partial \mathcal{O}_j}{\partial W_{ij}} \sum_{k \in K} (\mathcal{O}_k - \tau_k) \mathcal{O}_k (1 - \mathcal{O}_k) W_{jk} \\
&= \mathcal{O}_j (1 - \mathcal{O}_j) \frac{\partial x_j}{\partial W_{ij}} \sum_{k \in K} (\mathcal{O}_k - \tau_k) \mathcal{O}_k (1 - \mathcal{O}_k) W_{jk} \\
&= \mathcal{O}_j (1 - \mathcal{O}_j) \mathcal{O}_i \sum_{k \in K} (\mathcal{O}_k - \tau_k) \mathcal{O}_k (1 - \mathcal{O}_k) W_{jk} \\
&= \mathcal{O}_i \mathcal{O}_j (1 - \mathcal{O}_j) \sum_{k \in K} \delta_k W_{jk} \\
&= \mathcal{O}_i \phi_j
\end{aligned}$$

where we defined $\phi_j = \mathcal{O}_j(1 - \mathcal{O}_j) \sum_{k \in K} \delta_k W_{jk}$, which can be thought of as a weighted error at the output scaled by the derivative of the transfer function.

Because we can think of bias terms as connecting inputs always equal to 1, it is easy to verify that:

$$\frac{\partial E}{\partial \theta_l} = \begin{cases} \delta_l & \text{for } l \in K \\ \phi_l & \text{for } l \in J \end{cases}$$

The above derivation leads to the following algorithm for training the network from data:

- run the network forward on the inputs from the training set to compute its predictions
- for each output node compute $\delta_k = (\mathcal{O}_k - \tau_k)\mathcal{O}_k(1 - \mathcal{O}_k)$
- for each hidden node compute $\phi_j = \mathcal{O}_j(1 - \mathcal{O}_j) \sum_{k \in K} \delta_k W_{jk}$
- compute

$$\Delta_l = \begin{cases} \delta_l & \text{for } l \in K \\ \phi_l & \text{for } l \in J \end{cases}$$

- update:

$$W \leftarrow W_{ij} - \eta \mathcal{O}_i \Delta_j$$

$$\theta_l \leftarrow \theta_l - \eta \Delta_l$$

We normally run the algorithm on smaller batches of examples instead of the entire training set, in what is called episodes. After repeating the training for a number of episodes we save the resulting weights, and then we can use the network to make predictions outside of the training set. There are various criteria we can use to determine when to stop. A popular choice is to stop when the change in loss between episodes slows down to some value. The learning rate, batch size, stopping criteria, are all examples of hyper-parameters. These aren't learned but are rather usually set by experts using some industry heuristics. There is a significant research effort aimed at automating hyper-parameter selection, but in general it is a hard problem, and good hyper-parameter choices are part of the art of deep learning.

We have derived a basic version of stochastic gradient descent for optimization of loss function in neural networks known as back-propagation learning algorithm. There are many modifications to this basic approach used in modern deep learning research. Some of them are inspired by physics to simulate momentum of a rolling ball on the energy landscape defined by the loss function, and require computing higher order derivatives of the error. We will not go into detail on those extensions here, but a good overview is given in [91].