Notes on Neural Networks

Paulo Eduardo Rauber

1 Artificial neurons

Consider the data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) \mid i \in \{1, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m, \mathbf{y}_i \in \mathbb{R}^d\}$. The task of supervised learning consists on finding a function $f : \mathbb{R}^m \to \mathbb{R}^d$ that is able to *generalize* from the examples in $\mathcal{D}$. Supervised learning has been successful at image classification, natural language processing, and many other tasks.

Some types of artificial neural networks can be seen as a function $f : \mathbb{R}^m \to \mathbb{R}^d$ computed by a *network* of artificial neurons. The particular way in which the neurons are connected defines the *architecture* of the network. Artificial neurons are simplified models of real neurons. The objective of these models is not to simulate the behavior of real neurons, but to replicate some of their high level functionality.

Consider an input vector $\mathbf{x} = (x_1, \dots, x_m) \in \mathbb{R}^m$ to an artificial neuron. We define a weight vector $\mathbf{w} = (w_1, \dots, w_m) \in \mathbb{R}^m$, a bias $b \in \mathbb{R}$, and a threshold $\theta \in \mathbb{R}$. Using these definitions, the following expressions give the outputs of different models of artificial neurons.

- Linear neurons: $b + \sum_{i=1}^m x_i w_i. = b + \mathbf{xw}$.
- Rectified linear neurons: $\max(0, b + \mathbf{xw})$.
- Binary thresholded neurons: $\begin{cases} 1 & \text{if } b + \mathbf{xw} \geq \theta, \\ 0 & \text{otherwise.} \end{cases}$
- Sigmoid neurons: $\frac{1}{1+e^{-(b+\mathbf{xw})}}$.
- Stochastic binary neurons (sampling): $P(Y = 1 : \mathbf{w}, b) = \frac{1}{1+e^{-(b+\mathbf{xw})}}$.

The bias term b is the negative intercept of the affine hyperplane $H = \{(x_1, \dots, x_m) \mid x_1 w_1 + \ldots + x_m w_m = -b\}$. By consequence, binary thresholded neurons separate $\mathbb{R}^m$ into two half-spaces by an affine hyperplane. Sigmoid neurons perform an analogous but *smooth* assignment of the input $\mathbf{x}$ to one of the half-spaces.

Consider binary thresholded, sigmoid or stochastic binary neurons. Intuitively, a high bias indicates that a neuron is easy to excite (output a positive value). A low bias indicates the opposite. If each input element x_i is interpreted as a feature, a positive weight w_i indicates that the feature excites the neuron, and a negative weight w_i indicates that the feature inhibits the neuron.

Supervised learning in artificial neural networks is performed by updating the weight vectors of the artificial neurons to minimize a metric of prediction error on the training data set. When applied to discriminate between d classes, the artificial neural network may output d real values, which correspond to the confidence that the input belongs to each class.

2 Feedforward neural networks

This section describes feedforward neural networks, an important class of artificial neural networks.

Consider the data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) \mid i \in \{1, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m, \mathbf{y}_i \in \mathbb{R}^d\}$. Let $L \in \mathbb{N}^+$ represent the number of layers in the network, and $N^{(l)} \in \mathbb{N}^+$ represent the number of neurons in layer l, with $N^{(L)} = d$. We will refer to a neuron in layer l by a corresponding number between 1 and $N^{(l)}$. The neurons in the first layer are also called input units, the neurons in the output (last) layer called output units, and the other neurons called hidden units. Networks with more than 3 layers are called deep networks.

Let $w_{j,k}^{(l)} \in \mathbb{R}$ represent the weight reaching neuron j in layer l from neuron k in layer $(l-1)$. The order of the indices is counterintuitive, but makes the presentation simpler. Furthermore, let $b_j^{(l)} \in \mathbb{R}$ represent the bias for neuron j in layer l.

Consider a layer l, for $1 < l \leq L$, and neuron j, for $1 \leq j \leq N^{(l)}$. The weighted input to neuron j in layer l is defined as

$$z_j^{(l)} = b_j^{(l)} + \sum_{k=1}^{N^{(l-1)}} w_{j,k}^{(l)} a_k^{(l-1)},$$

where the activation of neuron j in layer $l > 1$ is defined as

$$a_j^{(l)} = \sigma(z_j^{(l)}),$$

where σ is a differentiable activation function. For example, consider the sigmoid activation function defined by $\sigma(z) = \frac{1}{1+e^{-z}}$. In this case, it is easy to show that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

It will be useful to define vectors (and a matrix) that represent quantities associated to each neuron in a given layer. The weighted input for layer $l > 1$ is defined as $\mathbf{z}^{(l)} = (z_1^{(l)}, \dots, z_{N^{(l)}}^{(l)})$, and the activation vector for layer l is defined as $\mathbf{a}^{(l)} = (a_1^{(l)}, \dots, a_{N^{(l)}}^{(l)})$. Furthermore, we define the bias vectors as $\mathbf{b}^{(l)} = (b_1^{(l)}, \dots, b_{N^{(l)}}^{(l)})$, and the weight matrices $W^{(l)}$ of dimension $N^{(l)} \times N^{(l-1)}$ as $(W^{(l)})_{j,k} = w_{j,k}^{(l)}$.

Using these definitions, the output of each layer $l > 1$ can be written as

$$\mathbf{a}^{(l)} = \sigma(W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}),$$

where the activation function is applied element-wise. Given these definitions, the output of the feedforward neural network f when $\mathbf{a}^{(1)} = \mathbf{x}$ is defined as the activation vector of the output layer $f(\mathbf{x}) = \mathbf{a}^{(L)}$. This completes the definition of the feedforward neural network model. It is possible to show that a feedforward neural network with a single hidden layer of sigmoid neurons can approximate any real-valued continuous function defined on a compact subset of $\mathbb{R}^m$.

Let the cost C be a differentiable function of the weights and biases. For example, consider the (halved) mean squared cost:

$$C = \frac{1}{n} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} c,$$

where

$$c = \frac{1}{2} ||\mathbf{a}^{(L)} - \mathbf{y}||^2,$$

when $\mathbf{a}^{(1)} = \mathbf{x}$. It is easy to show that $\frac{\partial c}{\partial a_j^{(L)}} = a_j^{(L)} - y_j$.

We define the task of learning the parameters (weights and biases) for a feedforward neural network as finding parameters that minimize C for a given training data set $\mathcal{D}$. The fact that the cost can be written as an average of costs for each element of the data set will be crucial to the proposed optimization procedure. The procedure requires the computation of partial derivatives of the cost with respect to weights and biases, which are usually computed by a technique called backpropagation.

Let the error of neuron j in layer l for a fixed $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ be defined as

$$\delta_j^{(l)} = \frac{\partial c}{\partial z_j^{(l)}}.$$

The error for the neurons in layer l is denoted by $\boldsymbol{\delta}^{(l)} = (\delta_1^{(l)}, \dots, \delta_{N^{(l)}}^{(l)})$.

We also let $\nabla_{\mathbf{a}} c = (\frac{\partial c}{\partial a_1^{(L)}}, \dots, \frac{\partial c}{\partial a_{N^{(L)}}^{(L)}})$ denote the gradient of c with respect to $\mathbf{a}^{(L)}$.

Backpropagation is a method for computing the partial derivatives of the cost function of a feedforward artificial neural network with respect to its parameters. The method is based solely on the following six statements, which we will demonstrate shortly:

$$\begin{aligned}
\boldsymbol{\delta}^{(L)} &= \nabla_{\mathbf{a}} c \odot \sigma'(\mathbf{z}^{(L)}), &(1)\\
\boldsymbol{\delta}^{(l)} &= ((W^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)}), &(2)\\
\frac{\partial c}{\partial b_j^{(l)}} &= \delta_j^{(l)}, &(3)\\
\frac{\partial c}{\partial w_{j,k}^{(l)}} &= a_k^{(l-1)} \delta_j^{(l)}, &(4)\\
\frac{\partial C}{\partial b_j^{(l)}} &= \frac{1}{n} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} \frac{\partial c}{\partial b_j^{(l)}}, &(5)\\
\frac{\partial C}{\partial w_{j,k}^{(l)}} &= \frac{1}{n} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} \frac{\partial c}{\partial w_{j,k}^{(l)}}, &(6)
\end{aligned}$$

where $\odot$ denotes element-wise multiplication. Notice how every quantity on the right side can be computed easily from our definitions, by starting with the errors in the output layer for every observation. That is why this method is called backpropagation.

The statement $\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{a}} c \odot \sigma'(\mathbf{z}^{(L)})$ is equivalent to

$$\delta_j^{(L)} = \frac{\partial c}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}),$$

for $1 \leq j \leq N^{(L)}$. By definition:

$$\delta_j^{(L)} = \frac{\partial c}{\partial z_j^{(L)}}.$$

First, notice that $a_j^{(L)}$ is a differentiable function of $z_j^{(L)}$, and c is a differentiable function of $a_1^{(L)}, \ldots, a_{N^{(L)}}^{(L)}$. Because $z_j^{(L)}$ only affects c through $a_j^{(L)}$, and $a_1^{(L)}, \ldots, a_{N^{(L)}}^{(L)}$ do not directly affect each other:

$$\delta_j^{(L)} = \frac{\partial c}{\partial z_j^{(L)}} = \sum_{k=1}^{N^{(L)}} \frac{\partial c}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_j^{(L)}} = \frac{\partial c}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial c}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}),$$

where the second equality comes from the chain rule, and the third equality from the fact that $\frac{\partial a_k^{(L)}}{\partial z_j^{(L)}} = 0$ for $k \neq j$. This completes the proof.

The statement $\boldsymbol{\delta}^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(\mathbf{z}^{(l)})$ is equivalent to

$$\delta_j^{(l)} = \sigma'(z_j^{(l)}) \sum_{k=1}^{N^{(l+1)}} w_{k,j}^{(l+1)} \delta_k^{(l+1)},$$

for $1 < l < L$ and $1 \leq j \leq N^{(l)}$.

Because $z_k^{(l+1)}$ is a differentiable function of $z_1^{(l)}, \ldots, z_{N^{(l)}}^{(l)}$, and c is a differentiable function of $z_1^{(l+1)}, \ldots, z_{N^{(l+1)}}^{(l+1)}$:

$$\delta_j^{(l)} = \frac{\partial c}{\partial z_j^{(l)}} = \sum_{k=1}^{N^{(l+1)}} \frac{\partial c}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}.$$

By definition,

$$z_k^{(l+1)} = b_k^{(l+1)} + \sum_{i=1}^{N^{(l)}} w_{k,i}^{(l+1)} a_i^{(l)},$$

therefore,

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \frac{\partial}{\partial z_j^{(l)}} \left[w_{k,j}^{(l+1)} a_j^{(l)} \right] = w_{k,j}^{(l+1)} \sigma'(z_j^{(l)}).$$

This gives

$$\delta_j^{(l)} = \sum_{k=1}^{N^{(l+1)}} \frac{\partial c}{\partial z_k^{(l+1)}} w_{k,j}^{(l+1)} \sigma'(z_j^{(l)}) = \sum_{k=1}^{N^{(l+1)}} \delta_k^{(l+1)} w_{k,j}^{(l+1)} \sigma'(z_j^{(l)}) = \sigma'(z_j^{(l)}) \sum_{k=1}^{N^{(l+1)}} w_{k,j}^{(l+1)} \delta_k^{(l+1)},$$

which completes the proof.

Consider the statement $\frac{\partial c}{\partial b_j^{(l)}} = \delta_j^{(l)}$. Because $z_j^{(l)}$ is a differentiable function of $b_j^{(l)}$ and c is a differentiable function of $z_1^{(l)}, \dots, z_{N^{(l)}}^{(l)}$:

$$\frac{\partial c}{\partial b_j^{(l)}} = \sum_{k=1}^{N^{(l)}} \frac{\partial c}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \frac{\partial c}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \frac{\partial c}{\partial z_j^{(l)}} = \delta_j^{(l)},$$

since $\frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = 0$ for $k \neq j$, and 1 otherwise. This completes the proof.

Similarly, consider the statement $\frac{\partial c}{\partial w_{j,k}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)}$. Because $z_j^{(l)}$ is a differentiable function of $w_{j,1}^{(l)}, \dots, w_{j,N^{(l-1)}}^{(l)}$, and c is a differentiable function of $z_1^{(l)}, \dots, z_{N^{(l)}}^{(l)}$:

$$\frac{\partial c}{\partial w_{j,k}} = \sum_{i=1}^{N^{(l)}} \frac{\partial c}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{j,k}} = \frac{\partial c}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{j,k}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial w_{j,k}},$$

since $\frac{\partial z_i^{(l)}}{\partial w_{j,k}} = 0$ if $i \neq j$. By definition, $z_j^{(l)} = b_j^{(l)} + \sum_{i=1}^{N^{(l-1)}} w_{j,i} a_i^{(l-1)}$. Therefore:

$$\frac{\partial z_j^{(l)}}{\partial w_{j,k}^{(l)}} = \frac{\partial}{\partial w_{j,k}^{(l)}} \left[w_{j,k}^{(l)} a_k^{(l-1)} \right] = a_k^{(l-1)},$$

which gives $\frac{\partial c}{\partial w_{j,k}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)}$, as we wanted to show.

The statements $\frac{\partial C}{\partial b_j^{(l)}} = \frac{1}{n} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} \frac{\partial c}{\partial b_j^{(l)}}$ and $\frac{\partial C}{\partial w_{j,k}^{(l)}} = \frac{1}{n} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}} \frac{\partial c}{\partial w_{j,k}^{(l)}}$ follow easily from the definition of C and c and their differentiability with respect to the parameters.

This completes the proof of the six statements that allow the computation of the gradient of C with respect to the parameters of the network through backpropagation. The statements are valid for other activation and cost functions, as we will show in the next Section. In each case, it is important to check whether the assumptions made in the proofs apply.

Intuitively, for each observation, backpropagation considers the effect of a small increase of Δw on a parameter w in the network. This change affects every subsequent neuron on a path to the output, and ultimately changes the cost c by a small Δc.

Consider a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ over variables $\mathbf{x} = (x_1, \dots x_n)$. Suppose we are interested in finding a global minimum of f. Because the gradient $\nabla f(\mathbf{a})$ gives the direction of maximum local increase in f at $\mathbf{a}$, it would be natural to start at a point $\mathbf{a}_0$, which could be chosen at random, and visit the sequence of points given by

$$\mathbf{a}_{t+1} = \mathbf{a}_t - \eta \nabla f(\mathbf{a}_t),$$

where the learning rate $\eta \in \mathbb{R}^+$ is a small constant. This technique is called gradient descent, and can be used as a heuristic to find the parameters that minimize the cost C with respect to the parameters of the network. Gradient descent is not guaranteed to converge. Even if it converges, the point at convergence may be a saddle point or a poor local minima. The choice of η considerably affects the success of gradient descent.

In a given iteration t of gradient descent, instead of computing $\frac{\partial C}{\partial w_{j,k}^{(l)}}$ and $\frac{\partial C}{\partial b_j^{(l)}}$ as averages derived from a computation involving all $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$, it is also possible to consider only a subset $\mathcal{D}' \subseteq \mathcal{D}$ of randomly chosen observations. The data set $\mathcal{D}$ may also be partitioned randomly into subsets called batches, which are considered in sequence. In this case, another random partition is considered once every subset is used. This procedure, called mini-batches stochastic gradient descent, is widely used due to its efficiency. Intuitively, the procedure makes faster decisions based on sampling. Regardless of these choices, a sequence of iterations that considers all observations in the data set is called an epoch.

The basic choices involved in learning the parameters for a feedforward neural network using stochastic gradient descent include at least the number of hidden layers, the number of neurons in each hidden layer, size of the mini-batches, the number of epochs, and the learning rate η. However, more choices will be necessary for the enhancements proposed in the next chapters.

3 Cost functions for feedforward neural networks

The previous section introduced feedforward neural networks. For simplicity, the presentation mentioned only sigmoid activation functions and the (halved) mean squared cost function. This section begins describes better alternatives for these functions.

Consider the weighted input $z_j^{(L)}$ to neuron j in the output layer L. If $|z_j^{(L)}|$ is large, the corresponding neuron is said to be saturated. In that case, if σ is the sigmoid function, then $\sigma'(z_j^{(L)})$ is close to zero. If the cost function is the mean squared cost, the first backpropagation statement gives $\delta_j^{(L)} = (a_j^{(L)} - y_j)\sigma'(z_j^{(L)})$. Therefore, in this case, $\delta_j^{(L)}$ is close to zero even if the target output y_j is very different from $a_j^{(L)}$. Another consequence is that the partial derivatives of the cost with respect to $w_{j,k}^{(L)}$ and $b_j^{(L)}$, for all k, will also be close to zero. This is expected from the mean squared cost, because a small change to the output, which requires a significant change to the parameters of a saturated neuron, would not improve the cost significantly. This has the undesired effect of slow learning for saturated neurons in the output layer. Since changing the learning rate would impact learning on the whole network, that is not an elegant solution.

Instead of considering the (halved) mean squared cost, suppose we considered a cost function c for a single observation such that $\frac{\partial c}{\partial a_j^{(L)}} = \frac{a_j^{(L)} - y_j}{\sigma'(z_j^{(L)})}$. The first backpropagation statement would give $\delta_j^{(L)} = a_j^{(L)} - y_j$, which would avoid slow learning for saturated neurons in the output layer. That is precisely why sigmoid output neurons are used together with the cross-entropy cost function.

Consider the data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) \mid i \in \{1, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m, \mathbf{y}_i \in \mathbb{R}^d\}$. The cross-entropy cost function c for a single pair $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ is defined as

$$c = -\sum_{i=1}^{d} \left[y_i \ln a_i^{(L)} + (1 - y_i) \ln (1 - a_i^{(L)}) \right],$$

when $\mathbf{a}^{(1)} = \mathbf{x}$. Notice that the cost function requires $a_i^{(L)} \in (0, 1]$ for all i. Intuitively, the cost is close to zero when $y_i \approx a_i^{(L)}$, and tends to infinity when $|a_i^{(L)} - y_i| \approx 1$.

Consider the partial derivative of the cost c with respect to $a_j^{(L)}$:

$$\frac{\partial c}{\partial a_j^{(L)}} = -\frac{y_j}{a_j^{(L)}} + \frac{(1 - y_j)}{(1 - a_j^{(L)})} = \frac{-y_j(1 - a_j^{(L)}) + a_j^{(L)}(1 - y_j)}{a_j^{(L)}(1 - a_j^{(L)})} = \frac{a_j^{(L)} - y_j}{a_j^{(L)}(1 - a_j^{(L)})}.$$

If $a_j^{(L)} = \sigma(z_j^{(L)})$, and σ is the sigmoid function, then $a_j^{(L)}(1 - a_j^{(L)}) = \sigma'(z_j^{(L)})$, and $\frac{\partial c}{\partial a_j^{(L)}} = \frac{a_j^{(L)} - y_j}{\sigma'(z_j^{(L)})}$. By the first backpropagation statement, $\delta_j^{(L)} = a_j^{(L)} - y_j$, as we wanted to show. As previously mentioned, the cross-entropy cost function avoids the slow down in learning caused by saturated sigmoid output neurons. However, that does not necessarily solve the analogous problem for the previous layers. In comparison with the previous section, the cross-entropy cost with sigmoid neurons only affects the computation of $\nabla_{\mathbf{a}} c$.

It is also common to use a different activation function in the output layer. Consider a network where the activation function for neuron j in the output layer L is is the identity $a_j^{(L)} = z_j^{(L)}$. Consider again the cost per pair $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ given by the (halved) mean squared cost

$$c = \frac{1}{2} ||\mathbf{a}^{(L)} - \mathbf{y}||^2,$$

when $\mathbf{a}^{(1)} = \mathbf{x}$. In this case, $\frac{\partial c}{\partial a_j^{(L)}} = a_j^{(L)} - y_j$. This change in the activation function for the last layer only affects the first backpropagation statement. From the definition of $\delta_j^{(L)}$, it is easy to show that

$$\delta_j^{(L)} = \sum_{k=1}^{N^{(L)}} \frac{\partial c}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_j^{(L)}} = \frac{\partial c}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = a_j^{(L)} - y_j.$$

The last equality follows from $\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = 1$. Therefore, the (halved) mean squared cost does not slow learning for saturated linear output neurons. This may be an appropriate choice for regression problems, which consist on predicting real-valued outputs. In comparison with the previous section, the (halved) mean square cost with linear output neurons only affects the first statement of backpropagation, which should be substituted by $\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{a}} c = \boldsymbol{a}^{(L)} - \boldsymbol{y}$.

Consider a data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) \mid i \in \{1, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m, \mathbf{y}_i \in \mathbb{R}^d\}$ for a d-classes classification problem. Concretely, for every pair $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$, $y_j = 1$ if $\mathbf{x}$ belongs to class j and $y_j = 0$ otherwise. The learning task can be restated as finding the parameters $\boldsymbol{\theta}$ that maximize the likelihood $L(\boldsymbol{\theta} : \mathcal{D})$ of observing $\mathbf{y}_i$ given $\mathbf{x}_i$, for every i, in a conditional probability distribution P parameterized by $\boldsymbol{\theta} \in \Theta$, assuming the sample elements in $\mathcal{D}$ are identically distributed and independent given any $\boldsymbol{\theta}$. The likelihood $L(\boldsymbol{\theta} : \mathcal{D})$ is defined as

$$L(\boldsymbol{\theta} : \mathcal{D}) = \prod_{i=1}^{n} P(\mathbf{Y} = \mathbf{y_i} \mid \mathbf{X} = \mathbf{x_i} : \boldsymbol{\theta}).$$

This is equivalent to finding the parameters $\boldsymbol{\theta}$ that minimize the negative log-likelihood $-\ell(\boldsymbol{\theta} : \mathcal{D})$, given by:

$$-\ell(\boldsymbol{\theta} : \mathcal{D}) = -\sum_{i=1}^{n} \ln P(\mathbf{Y} = \mathbf{y_i} \mid \mathbf{X} = \mathbf{x_i} : \boldsymbol{\theta}).$$

Suppose we could interpret the output $a_j^{(L)}$ of neuron j in the output layer L of a feedforward neural network as the probability of input $\mathbf{a}^{(1)} = \mathbf{x}$ belonging to class j. The negative log-likelihood cost function c for a pair $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ could be written as

$$c = -\sum_{i=1}^{d} y_i \ln a_i^{(L)},$$

where $\mathbf{a}^{(1)} = \mathbf{x}$. Notice that this cost function requires $a_j^{(L)} > 0$.

The softmax output layer is used precisely so the output of the network can be interpreted as a parameterized conditional probability distribution $P(\mathbf{Y} \mid \mathbf{X} : \boldsymbol{\theta})$. A feedforward neural network has a softmax output layer when

$$a_j^{(L)} = \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{d} e^{z_k^{(L)}}},$$

for every neuron $1 \leq j \leq d$ in the output layer. It is easy to see that $\sum_{j=1}^{d} a_j^{(L)} = 1$ and $a_j^{(L)} > 0$ for all j. Given input $\mathbf{a}^{(1)} = \mathbf{x}$, the activation $a_j^{(L)}$ can be interpreted as $P(\mathbf{Y} = \mathbf{e}_j \mid \mathbf{X} = \mathbf{x} : \boldsymbol{\theta})$, where $\mathbf{e}_j$ is the standard basis vector in direction j, and $\boldsymbol{\theta}$ represents the parameters of the network.

The softmax activation function is significantly different from those presented so far. Notice that $a_j^{(L)}$ depends on $z_i^{(L)}$ for all i. In fact, this completely invalidates the first backpropagation statement. However, we will now show that

$$\boldsymbol{\delta}^{(L)} = \boldsymbol{a}^{(L)} - \boldsymbol{y}.$$

By definition, $\delta_j^{(L)} = \frac{\partial c}{\partial z_j^{(L)}}$. Consider the negative log-likelihood cost function $c = -\sum_{i=1}^{d} y_i \ln a_i^{(L)}$. Because c is a differentiable function of $a_1^{(L)}, \dots, a_d^{(L)}$, and $a_k^{(L)}$ is a differentiable function of $z_j^{(L)}$ for every j and k,

$$\frac{\partial c}{\partial z_j^{(L)}} = \sum_{k=1}^{d} \frac{\partial c}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_j^{(L)}}.$$

It is also easy to show that $\frac{\partial c}{\partial a_k^{(L)}} = -\frac{y_k}{a_k^{(L)}}$. Therefore:

$$\frac{\partial c}{\partial z_j^{(L)}} = \sum_{k=1}^{d} -\frac{y_k}{a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_j^{(L)}}.$$

Because the softmax output layer will be applied solely in a classification task, y_h is 1 for a single h, and zero otherwise. Therefore,

$$\frac{\partial c}{\partial z_j^{(L)}} = -\frac{1}{a_h^{(L)}} \frac{\partial a_h^{(L)}}{\partial z_j^{(L)}}.$$

Consider the derivative of the activation of neuron h with respect to the weighted input to neuron j:

$$\begin{aligned}\frac{\partial a_h^{(L)}}{\partial z_j^{(L)}} &= \frac{\partial}{\partial z_j^{(L)}} \left[\frac{e^{z_h^{(L)}}}{\sum_{k=1}^{d} e^{z_k^{(L)}}} \right] \\ &= \frac{1}{\left[\sum_{k=1}^{d} e^{z_k^{(L)}}\right]^2} \left[\frac{d}{dz_j^{(L)}} \left[e^{z_h^{(L)}}\right] \sum_{k=1}^{d} e^{z_k^{(L)}} - e^{z_h^{(L)}} e^{z_j^{(L)}} \right].\end{aligned}$$

By separating fraction above into two, and using the definition of the activations $a_h^{(L)}$ and $a_j^{(L)}$:

$$\frac{\partial a_h^{(L)}}{\partial z_j^{(L)}} = \frac{\frac{d}{dz_j^{(L)}} \left[e^{z_h^{(L)}}\right]}{\sum_{k=1}^{d} e^{z_k^{(L)}}} - a_h^{(L)} a_j^{(L)}.$$

Consider $\frac{d}{dz_j^{(L)}} \left[e^{z_h^{(L)}}\right]$. If $h = j$, then $\frac{d}{dz_j^{(L)}} \left[e^{z_h^{(L)}}\right] = e^{z_h^{(L)}}$. Otherwise, $\frac{d}{dz_j^{(L)}} \left[e^{z_h^{(L)}}\right] = 0$. Therefore:

$$\frac{\partial a_h^{(L)}}{\partial z_j^{(L)}} = \begin{cases} a_h^{(L)}(1 - a_j^{(L)}) & \text{if } h = j, \\ -a_h^{(L)} a_j^{(L)} & \text{otherwise.} \end{cases}$$

Consider again the equation

$$\frac{\partial c}{\partial z_j^{(L)}} = -\frac{1}{a_h^{(L)}} \frac{\partial a_h^{(L)}}{\partial z_j^{(L)}}.$$

Using the expression for $\frac{\partial a_h^{(L)}}{\partial z_j^{(L)}}$,

$$\frac{\partial c}{\partial z_j^{(L)}} = \begin{cases} a_j^{(L)} - 1 & \text{if } h = j, \\ a_j^{(L)} & \text{otherwise.} \end{cases}$$

The equation above can also be written as

$$\frac{\partial c}{\partial z_j^{(L)}} = a_j^{(L)} - y_j,$$

as we wanted to show.

In comparison with the previous section, the softmax output layer with negative log-likelihood cost only affects the first statement of backpropagation, which should be substituted by $\boldsymbol{\delta}^{(L)} = \boldsymbol{a}^{(L)} - \boldsymbol{y}$. This combination of cost function and output activation is a principled and elegant formulation of the classification task, and is often used in practice.

4 Improving feedforward neural networks

This section begins by describing techniques to deal with overfitting. In a feedforward neural network, learning consists on finding parameters (weights and biases) that minimize a cost function defined on the available data. In this context, the network is also called a model for the data. A model is said to overfit when it has a low cost over the available data, but generalizes poorly for unseen data.

The most common way to diagnose overfitting is to perform cross-validation. In cross-validation, the available data is partitioned into two sets. One of these sets is used for learning (training set), and the other set is used for evaluation (test set). This is also how model efficacy is evaluated.

However, feedforward neural networks also have hyperparameters. These are the parameters that must be defined even before learning occurs, which include the learning rate, number of layers, number of neurons in each layer, number of epochs, mini-batch size, etc. Evaluating generalization efficacy through hyperparameters chosen according to their efficacy on the test set is a methodological mistake. The hyperparameters may also overfit the test data, which would result in overly optimistic evaluations.

The available data is usually split into three sets: training, validation, and testing. The validation set is used to compare the results of different hyperparameters, which are used for learning using the training set. The best hyperparameters on the validation set are used for learning using both training and validation sets. Finally, the model is evaluated in the test set, which gives a generalization efficacy estimate. This process may be repeated several times, using schemes as k-fold cross-validation or bootstrapping, which we do not detail here.

In the case of feedforward neural networks, it is common to create graphs comparing cost and accuracy on training and validation data as epochs pass. This may help in diagnosing overfitting. For example, the accuracy on the training data may become perfect after a number of epochs, while the accuracy on the validation data becomes worse. Early stopping consists on halting the learning procedure when validation accuracy does not improve significantly after a given number of epochs.

Choosing hyperparameters for a feedforward neural network that achieve good generalization in the validation set can be a very challenging task. Because the number of hyperparameters is very large, manual fine-tuning is often preferred to schemes such as grid search.

Regularization is an important technique to prevent overfitting. Intuitively, regularization penalizes large weights in the network, which could make the output very sensitive to small changes in the input. We define the L2-regularized cost function C for a data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) \mid i \in \{1, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m, \mathbf{y}_i \in \mathbb{R}^d\}$ as

$$C = \frac{1}{n}\left[\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} c\right] + \frac{\lambda}{2n}\left[\sum_{w\in\mathbf{W}} w^2\right],$$

where c is the cost for the pair $(\mathbf{x}, \mathbf{y})$ when $\mathbf{a}^{(1)} = \mathbf{x}$, $\mathbf{W}$ is the set of all weights in the network, and $\lambda \geq 0$ is the regularization factor. Notice how the second term increases the overall cost of having large weights when the first term is fixed. Intuitively, in the case of an extremely large λ, the network would be rewarded for effectively removing most weights. It is important to note that regularization is not considered into the cost c for a single observation, because that would invalidate the fourth property of backpropagation.

It is easy to show that the partial derivative of the cost C with respect to bias $b_j^{(L)}$ is

$$\frac{\partial C}{\partial b_j^{(L)}} = \frac{1}{n}\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \frac{\partial c}{\partial b_j^{(L)}},$$

which is the same as before. The reason for not penalizing the biases is mostly empirical.

The partial derivative of the cost C with respect to weight $w_{j,k}^{(L)}$ is

$$\frac{\partial C}{\partial w_{j,k}^{(L)}} = \frac{1}{n}\left[\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \frac{\partial c}{\partial w_{j,k}^{(L)}}\right] + \frac{\lambda}{n} w_{j,k}^{(L)}.$$

When using gradient descent to minimize C with respect to the parameters of the network, the update rule for $w_{j,k}^{(L)}$ can be written as

$$\begin{aligned} w_{j,k}^{(L)} &\leftarrow w_{j,k}^{(L)} - \eta\Big[\frac{\partial C}{\partial w_{j,k}^{(L)}}\Big] \\ &= w_{j,k}^{(L)} - \eta\left[\frac{1}{n}\left[\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \frac{\partial c}{\partial w_{j,k}^{(L)}}\right] + \frac{\lambda}{n} w_{j,k}^{(L)}\right] \\ &= (1 - \frac{\eta\lambda}{n}) w_{j,k}^{(L)} - \eta\left[\frac{1}{n}\sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \frac{\partial c}{\partial w_{j,k}^{(L)}}\right]. \end{aligned}$$

Notice that the last rule is very similar to gradient descent on a cost function without regularization, except for the factor $0 \leq (1 - \frac{\eta\lambda}{n}) < 1$ (under sensible choices of η and λ) multiplying $w_{j,k}^{(L)}$. This regularization method is also called weight decay. When performing stochastic gradient descent, the decay factor is always $(1 - \frac{\eta\lambda}{n})$ for every update, irrespective of batch size.

Another way to reduce overfitting is to use more training data. Intuitively, this is useful to constrain the *freedom* in the choice of parameters, which is one of the major causes of overfitting. Because it is not always easy to obtain more data, one particularly useful idea is to expand the available data by transforming input vectors in ways that mimic expected variations. For example, if the input vectors correspond to images, transformations such as translation, scaling and rotation often should not affect classification.

Previously, we mentioned that stochastic gradient descent may begin at a random assignment to the weights and biases. Consider the neuron j in layer l, and let $p = N^{(l-1)}$ denote the number of neurons in layer $l - 1$, which is also the number of inputs to neuron j. Suppose the weights reaching neuron j were sampled according to a Gaussian distribution with mean 0 and standard deviation σ, such that $w_{j,k}^{(l)} \sim \mathcal{N}(0, \sigma^2)$, for all k. By the properties of Gaussian distributed random variables,

$$\sum_{k=1}^{p} w_{j,k}^{(l)} \sim \mathcal{N}(0, p\sigma^2).$$

If the weights were sampled from a standard Gaussian distribution $\mathcal{N}(0, 1)$, then $\sum_{k=1}^{p} w_{j,k}^{(l)} \sim \mathcal{N}(0, p)$. If the number of inputs p to neuron j were large, the neuron could easily become saturated. As already mentioned, saturated neurons slow learning, because small changes to their parameters do not significantly affect their outputs. In contrast, consider $w_{j,k}^{(l)} \sim \mathcal{N}(0, \frac{1}{p})$. In this case, $\sum_{k=1}^{p} w_{j,k}^{(l)} \sim \mathcal{N}(0, 1)$, which helps in avoiding early saturation and overall learning success. This is a common recommendation for weight initialization in feedforward neural networks. The bias $b_j^{(l)}$ for neuron j can be sampled from $\mathcal{N}(0, 1)$.

Consider a feedforward neural network with a single sigmoid neuron in each layer and a cross-entropy cost function. In this case, it is easy to show that the error $\delta_1^{(l)}$ of the neuron at layer l for a single pair $(\mathbf{x}, \mathbf{y})$ is

$$\delta_1^{(l)} = (a_1^{(L)} - y_1) \prod_{i=l}^{L-1} w_{1,1}^{(i+1)} \sigma'(z_1^{(i)}),$$

for every $l > 1$. If the weights in the network are not very large, a single saturated neuron could make $\delta_1^{(l)}$ very small. Thus, the partial derivatives of the parameters with respect to the cost would also be small in layer l, which would compromise learning by gradient descent in all layers previous to and including l. An analogous problem occurs in more general networks. Because c can be seen as a function of the weighted inputs $z_j^{(l)}$ for all neurons j in a given layer l, this issue is characterized by $||\nabla_{\mathbf{z}^{(l)}} c|| \approx 0$, which is why this learning problem is called vanishing gradients. When the weights in the network are large, *exploding* (large) gradients could also become a problem. However, large weights are often followed by saturated neurons, which makes the problem less common. It may also be important to scale input vector elements to have zero mean and unit variance across every input vector to avoid early saturation.

It can be shown that deep feedforward neural networks ($L > 3$ layers) can approximate more *complicated* functions with fewer parameters than shallower networks. However, in practice, vanishing gradients make it difficult to obtain better efficacy with deep feedforward neural networks trained by gradient descent. We now describe two techniques considered important in the context of deep feedforward neural networks: momentum and dropout.

The momentum technique is a common heuristic for training deep artificial neural networks. In momentum-based stochastic gradient descent, each parameter w in the network (weight or bias) has a corresponding velocity v. The velocity is defined by $v_0 = 0$ and

$$v_{t+1} = \mu v_t - \eta \left[\frac{\partial C}{\partial w_t} \right],$$

where v_i and w_i correspond, respectively, to v and w at iteration i of stochastic gradient descent. At each iteration, the parameter w is updated by the rule $w_{t+1} = w_t + v_{t+1}$. Intuitively, the momentum technique remembers the velocity of each parameter, allowing larger updates when the direction of decrease in cost is consistent over many iterations. The parameter $0 \leq \mu \leq 1$ controls the effect of the previous velocity on the next velocity, and $1 - \mu$ is commonly interpreted as a coefficient of friction. If $\mu = 0$, the technique is equivalent to stochastic gradient descent.

Dropout is another heuristic for training deep artificial neural networks. At every iteration of stochastic gradient descent, *half* the hidden neurons are removed at random. In most implementations, this can be accomplished by forcing the outputs of the corresponding neurons to be zero. The modified network is applied as usual to the observations in a mini-batch, and backpropagation follows, as if the network were not changed. The resulting partial derivatives are used to update the parameters of the neurons that were not removed. After training is finished, the weights incoming from hidden neurons are *halved*. This heuristic is believed to make the network robust to the absence of particular features, which might be particular to the training data. Dropout is considered related to regularization for trying to reduce overfitting.

There are many heuristics for implementing feedforward neural networks that will not be described in detail in this text. They include Hessian optimization, input whitening, rmsprop, and adaptive learning rates. Although we focused most of the discussion on sigmoid neurons, rectified linear neurons have achieved superior results in important benchmarks.

5 Convolutional neural networks

Convolutional neural networks were first developed for image classification, although they have also been applied in other tasks. This section focuses on image classification using convolutional neural networks.

A two-dimensional image is a function $f : \mathbb{Z}^2 \to \mathbb{R}^c$. If $f(\mathbf{a}) = (f_1(\mathbf{a}), \dots, f_c(\mathbf{a}))$, then f_i is also called channel i. An element $\mathbf{a} \in \mathbb{Z}^2$ is called a pixel, and $f(\mathbf{a})$ is the value of pixel $\mathbf{a}$. A window $W \subset \mathbb{Z}^2$ is a finite set $W = [s_1, S_1] \times [s_2, S_2]$ that corresponds to a rectangle in the image domain. The size of this window W is denoted as $w \times h$, where $w = S_1 - s_1 + 1$ and $h = S_2 - s_2 + 1$. Because the domain D of images of interest is usually a window, it is possible to represent an image f by a vector $\mathbf{x} \in \mathbb{R}^{c|D|}$. In this vector, there is a scalar value $f_i(\mathbf{a})$ corresponding to each channel i of each pixel $\mathbf{a} \in D$.

Consider a data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) \mid i \in \{1, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m, \mathbf{y}_i \in \mathbb{R}^d\}$, where each vector $\mathbf{x}_i$ corresponds to a distinct image $f : \mathbb{Z}^2 \to \mathbb{R}^c$, all images are defined on the same window D, and $m = c|D|$. The task of image classification consists on assigning a class label for a given vector $\mathbf{x}$ based on generalization from $\mathcal{D}$.

A convolutional neural network is particularly well suited for image classification, because it explores the spatial relationships between pixels (organization in $\mathbb{Z}^2$). Similarly to feedforward neural networks, a convolutional neural network is also a parameterized function, and the parameters are usually learned by stochastic gradient descent on a cost function defined on the training set. In contrast to feedforward neural networks, there are three main types of layers in a convolutional neural network: convolutional layers, pooling layers and fully connected layers.

A convolutional layer receives an input image f and outputs an image o. A convolutional layer is composed solely of artificial neurons. Each artificial neuron h in a convolutional layer l receives as input the values in a window $W = [s_1, S_1] \times [s_2, S_2] \subset D$ of size $w \times h$, where D is the domain of f. The weighted output $z_h^{(l)}$ of that neuron is given by

$$z_h^{(l)} = b_h^{(l)} + \sum_{i=1}^{c} \sum_{j=s_1}^{S_1} \sum_{k=s_2}^{S_2} w_{h,i,j,k}^{(l)} a_{i,j,k}^{(l-1)}.$$

In the equation above, $a_{i,j,k}^{(l-1)} = f_i(j,k)$, the value of pixel (j,k) in channel i of the input image. Also, $b_h^{(l)}$ is the bias of neuron h and $w_{h,i,j,k}^{(l)}$ the weight that neuron h in layer l associates to $f_i(j,k)$. The activation function for a convolutional layer is usually rectified linear, so $a_h^{(l)} = \max(0, z_h^{(l)})$. The definition of $z_h^{(l)}$ is similar to the definition of the weighted input for a neuron in a feedforward neural network. The only difference is that a neuron in a convolutional layer is not necessarily connected to the activations of all neurons in the previous layer, but only to the activations in a particular $w \times h$ window W. Each neuron in a convolutional layer has cwh weights and a single bias.

A neuron in a convolutional layer is replicated (through parameter *sharing*) for all windows of size $w \times h$ in the domain D whose centers are offset by pre-defined steps. These steps are the horizontal and vertical *strides*. The activations corresponding to a neuron replicated in this way correspond to the values in a single channel of the output image o (appropriately arranged in $\mathbb{Z}^2$). Thus, an output image $o : \mathbb{Z}^2 \to \mathbb{R}^N$ is obtained by replicating N neurons over the whole domain of the input image. The total number of free parameters in a convolutional layer is only $N(cwh + 1)$. If the parameters in a convolutional layer were not shared by replicated neurons, the number of parameters would be $MN(cwh + 1)$, where M is the number of windows of size $w \times h$ that fit into f (for the given strides).

The discrete convolution of an image $f : \mathbb{Z}^2 \to \mathbb{R}^c$ with an image $g : \mathbb{Z}^2 \to \mathbb{R}^c$ is a function $f * g$ defined as

$$(f * g)(n, m) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j) \cdot g(i - n, j - m).$$

Intuitively, the discrete convolution of f and g at (n, m) corresponds to the result of *sliding* g by (n, m), computing the inner product between the values of the two functions at every pixel, and adding all the resulting values. This operation can be implemented very efficiently for images defined on a window domain D.

The weighted outputs (minus the bias) of replicated neurons correspond to an output channel that is precisely the discrete convolution of the input f with a particular image g. The values of g correspond to the (shared) weights of the replicated neurons (appropriately arranged in $\mathbb{Z}^2$). This assumes that the horizontal and vertical strides are 1 and that the domain of $f * g$ is always restricted to the window domain of f. In other words, each channel o_u in the output o of a convolutional layer corresponds to a convolution with an image g_u, which is also called a filter. This is the origin of the name convolutional network. Therefore, to define a convolutional layer, it is enough to specify the size of the filters (window size), the number of filters (number of channels in the output image), horizontal and vertical strides (which are usually 1).

Each channel in the output of a convolutional layer can also be seen as the response of the input image to a particular (learned) filter. Based on this interpretation, each channel in the output image is also called an activation map.

Backpropagation can be adapted to compute the partial derivative of the cost with respect to every parameter in a convolutional layer. The fact that a single weight affects the output of several neurons must be taken into account. We omit the details of backpropagation for convolutional neural networks in this text.

In summary, a convolutional layer receives an input image $f : \mathbb{Z}^2 \to \mathbb{R}^c$ and outputs an image $o : \mathbb{Z}^2 \to \mathbb{R}^N$ defined by

$$o_i(\mathbf{a}) = \sigma\Big[(f * g_i)(\mathbf{a}) + b_i\Big],$$

where $i \in \{1, \ldots, N\}$, $\mathbf{a} \in \mathbb{Z}^2$ is a pixel, $g_i : \mathbb{Z}^2 \to \mathbb{R}^c$ is a filter image, $b_i \in \mathbb{R}$ is the bias for filter i, and σ is an activation function. As already mentioned, the equation above assumes that the horizontal and vertical strides are 1 and that the domain of $f * g$ is always restricted to the window domain of f

A pooling layer receives an input image $f : \mathbb{Z}^2 \to \mathbb{R}^c$ and outputs an image $o : \mathbb{Z}^2 \to \mathbb{R}^c$. A pooling layer reduces the size of the window domain D of f by an operation that acts independently on each channel. A common pooling technique is max-pooling. In max-pooling, the maximum value of channel f_i in a particular window of size $w \times h$ corresponds to an output value in channel o_i. To define a max-pooling layer, it is enough to specify the size of these windows and the strides (which usually match the window dimensions). The objective of reducing the spatial domain of the image is to achieve similar results to using comparatively larger convolutional filters in the next layers. This supposedly allows the detection of higher-level features in the input image with a reduced number of parameters. It is also believed that max-pooling improves the invariance of the classification to translations of the original image. In practice, a sequence of alternating convolutional and max-pooling layers has obtained excellent results in many image classification tasks. Backpropagation can also be performed through max-pooling layers.

In summary, a max-pooling layer receives an input image $f : \mathbb{Z}^2 \to \mathbb{R}^c$ and outputs an image $o : \mathbb{Z}^2 \to \mathbb{R}^c$ defined by

$$o_i(j, k) = \max_{\mathbf{a} \in W_{j,k}} f_i(\mathbf{a}),$$

where $i \in \{1, \ldots, c\}$, $(j, k) \in \mathbb{Z}^2$, D is the window domain of f, and $W_{j,k} \subseteq D$ is the input window corresponding to output pixel (j, k).

A fully connected layer receives an input image $f : \mathbb{Z}^2 \to \mathbb{R}^c$ or an input vector $\mathbf{x}$ and outputs a vector $\mathbf{o}$. A fully connected layer is precisely analogous to a layer in a feedforward neural network, and can only be succeeded by other fully connected layers. The final layer in a convolutional neural network is always a fully connected layer with d neurons, which is responsible for representing the classification. Backpropagation in fully connected layers is analogous to backpropagation in feedforward networks.

The task of detection corresponds to delimiting the spatial extent of particular objects of interest *within* an image. Consider a convolutional neural network trained on images with a window domain D. The input to the first fully connected layer is an image with a window domain W of size $w \times h$, which is dependent on the number of pooling layers present in the network. This image with domain W is ultimately mapped (through forward pass through fully connected layers) to a vector $\mathbf{o} \in \mathbb{R}^d$, which represents the confidence that the input image belongs to each class. Notice that it is possible to apply the same network, up to the first fully connected layer, to an input image with a much larger window domain D_L. This results in a larger input image with domain W_L for the first fully connected layer. It is possible to partition the window W_L into smaller windows W_l of size $w \times h$, which matches the size required by the first fully connected layer. Therefore, each smaller window W_l of this larger input image can be associated to an output vector $\mathbf{o}_l \in \mathbb{R}^d$, which represents the confidence that the input image contains a particular object in window W_l. As a consequence, the original larger image is partitioned into rectangular windows for which the confidence that each window contains a particular object is known. This partition can be represented by an image $g : \mathbb{Z}^2 \to \mathbb{R}^d$ called class map. Precisely the same class map would be obtained by exhaustively applying the convolutional network for each window of size $w \times h$ which fits into the larger input image, although the computational cost would be significantly higher.

In summary, for the purpose of detection, the convolutional neural network can be trained with fixed-size images of objects of interest. Detection is performed by resizing the input image so that the expected size of the objects of interest matches the size observed during training. The resized image is received as input by the adapted network, which partitions the input to the fully connected layers appropriately. The corresponding class map can be post-processed to obtain rectangular boundaries containing objects or confidences for the presence of a given object.

The choice of hyperparameters for convolutional neural networks (types of layers, number of layers, filters per layer, filter sizes, strides) is crucial to achieve good performance. Deep convolutional neural networks are usually trained in immense data sets, requiring (non-trivial) efficient implementations, which include all the improvements mentioned in the previous section. After training a convolutional neural network for a particular data set, it is possible to re-use the parameters of the network (up to its last fully connected layer) as a starting point for another classification task. This technique decouples *representation learning* from a specific image classification problem, and has been very successful in practice.

6 Recurrent neural networks

Let A^+ denote the set of non-empty sequences of elements in the set A, and let $|X|$ denote the length of a sequence $X \in A^+$. We denote the t-th element of sequence X by $X[t]$. Consider the data set $\mathcal{D} = \{(X_i, Y_i) \mid i \in \{1, \dots, n\}, X_i \in (\mathbb{R}^m)^+, Y_i \in (\mathbb{R}^d)^+\}$. Furthermore, let $|X| = |Y|$ for every $(X, Y) \in \mathcal{D}$. In other words, the data set $\mathcal{D}$ is composed of pairs (X, Y) of sequences of the same length. Each element of the two sequences is a real vector, but $X[t]$ and $Y[t]$ do not necessarily have the same dimension. The task of sequence element classification consists on finding a function $f : (\mathbb{R}^m)^+ \to (\mathbb{R}^d)^+$ that is able to *generalize* from the examples in $\mathcal{D}$. This task is more general than sequence classification, which consists on finding a function $f : (\mathbb{R}^m)^+ \to \mathbb{R}^d$. It is not straightforward to apply the models presented so far to sequence element classification.

Recurrent neural networks compose a particular class of artificial neural networks that can be applied to sequence element classification. We start by introducing a simple recurrent neural network with a single hidden layer. Many definitions are analogous to those presented for feedforward neural networks, and will be omitted when there is no ambiguity.

Let $N^{(l)}$ be the number of neurons in layer l, with $N^{(1)} = m$ and $N^{(3)} = d$. Let $b_j^{(l)} \in \mathbb{R}$ represent the bias for neuron j in layer $l > 1$, and $w_{j,k}^{(l)} \in \mathbb{R}$ represent the weight reaching neuron j in layer $l > 1$ from neuron k in layer $(l-1)$. Furthermore, let $\omega_{j,k}^{(2)} \in \mathbb{R}$ represent the weight reaching neuron j in layer 2 from neuron k in layer 2 from the previous *time step*.

The input activation at time t for $(X, Y) \in \mathcal{D}$ is defined as $\mathbf{a}[t]^{(1)} = X[t]$. The weighted input to neuron $1 \leq j \leq N^{(2)}$ in layer 2 at time step t is defined as

$$z[t]_j^{(2)} = b_j^{(2)} + \sum_{k=1}^{N^{(1)}} w_{j,k}^{(2)} a[t]_k^{(1)} + \sum_{k=1}^{N^{(2)}} \omega_{j,k}^{(2)} a[t-1]_k^{(2)},$$

where $\mathbf{a}[0]^{(2)}$ may be zero, although it could also be learned.

The activation of neuron j in a layer $l > 1$ at time t is defined as $a[t]_j^{(l)} = \sigma(z[t]_j^{(l)})$, where σ is a differentiable activation function. For concreteness, we will consider sigmoid activation functions.

The weighted input to neuron $1 \leq j \leq N^{(3)}$ in layer 3 is defined as

$$z[t]_j^{(3)} = b_j^{(3)} + \sum_{k=1}^{N^{(2)}} w_{j,k}^{(3)} a[t]_k^{(2)},$$

as in a feedforward neural network.

The output of the recurrent neural network f on input $X = \mathbf{a}[1]^{(1)}, \dots, \mathbf{a}[T]^{(1)}$ is the sequence $\mathbf{a}[1]^{(3)}, \dots, \mathbf{a}[T]^{(3)}$. This completes the definition of a recurrent neural network with a single recurrent hidden layer.

Intuitively, the sequence X is presented to the network element by element. The network behaves similarly to a single hidden layer feedforward neural network, except for the fact that the output activation $\mathbf{a}[t]^{(2)}$ of the hidden layer at time t possibly affects the weighted input $\mathbf{z}[t+1]^{(2)}$ of the hidden layer at time $t+1$. Intuitively, an ideal recurrent neural network would be capable of representing a sequence $X[1:t]$ by its hidden layer activation $\mathbf{a}[t]^{(2)}$ to allow correct classification of $X[t+1]$.

Once again, we define the task of learning the parameters (weights and biases) for a neural network as finding parameters that minimize a cost function $C = \frac{1}{n} \sum_{(X,Y) \in \mathcal{D}} c$. Several choices of c are possible, as discussed in a previous section. Concretely, consider the case of a cross-entropy cost function c for sequence element classification defined as

$$c = -\frac{1}{T} \sum_{t=1}^{T} \sum_{i=1}^{d} \Big[Y[t]_i \ln{(a[t]_i^{(3)})} + (1 - Y[t]_i) \ln{(1 - a[t]_i^{(3)})} \Big],$$

where $X = \mathbf{a}[1]^{(1)}, \dots, \mathbf{a}[T]^{(1)}$ is the input sequence and $\mathbf{a}[1]^{(3)}, \dots, \mathbf{a}[T]^{(3)}$ is the output sequence. As mentioned in a previous section, this cost function requires that every $Y[t]$ be a standard basis vector, which represents the fact that every $X[t]$ belongs to a single class.

The error of neuron j in layer l at time step t is defined as

$$\delta[t]_j^{(l)} = \frac{\partial c}{\partial z[t]_j^{(l)}}.$$

Backpropagation through time is a method for computing the partial derivatives of the cost function of a recurrent neural network with respect to its parameters. In the case of a recurrent neural network with a single hidden layer, the method depends solely on the following statements, which will be demonstrated shortly:

$$\delta[t]_j^{(3)} = \frac{\partial c}{\partial a[t]_j^{(3)}} \sigma'(z[t]_j^{(3)}), \tag{1}$$

$$\delta[t]_j^{(2)} = \sigma'(z[t]_j^{(2)}) \Big[\sum_{k=1}^{N^{(3)}} w_{k,j}^{(3)} \delta[t]_k^{(3)} + \sum_{k=1}^{N^{(2)}} \omega_{k,j}^{(2)} \delta[t+1]_k^{(2)} \Big], \tag{2}$$

$$\frac{\partial c}{\partial b_j^{(l)}} = \sum_{t=1}^{T} \delta[t]_j^{(l)}, \tag{3}$$

$$\frac{\partial c}{\partial w_{j,k}^{(l)}} = \sum_{t=1}^{T} \delta[t]_j^{(l)} a[t]_k^{(l-1)}, \tag{4}$$

$$\frac{\partial c}{\partial \omega_{j,k}^{(2)}} = \sum_{t=1}^{T} \delta[t]_j^{(2)} a[t-1]_k^{(2)}, \tag{5}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{1}{n} \sum_{(X,Y) \in \mathcal{D}} \frac{\partial c}{\partial b_j^{(l)}}, \tag{6}$$

$$\frac{\partial C}{\partial w_{j,k}^{(l)}} = \frac{1}{n} \sum_{(X,Y) \in \mathcal{D}} \frac{\partial c}{\partial w_{j,k}^{(l)}}, \tag{7}$$

$$\frac{\partial C}{\partial \omega_{j,k}^{(2)}} = \frac{1}{n} \sum_{(X,Y) \in \mathcal{D}} \frac{\partial c}{\partial \omega_{j,k}^{(2)}}, \tag{8}$$

where $\boldsymbol{\delta}[T+1]^{(2)}$ is zero. Notice how the error $\delta[t]_j^{(2)}$ depends on the error $\delta[t+1]_k^{(2)}$, for all j, k and t. Every quantity on the right side can be computed easily from our definitions by starting from $\boldsymbol{\delta}[T]^{(3)}$, which can only be computed after the network observes the entire sequence. That is why the method is called backpropagation through time.

A single hidden layer recurrent neural network can be *unfolded* into a feedforward network, which may be useful to interpret backpropagation through time. For each pair $(X, Y) \in \mathcal{D}$, let $X = \mathbf{a}[1]^{(1)}, \dots, \mathbf{a}[T]^{(1)}$, and consider a feedforward network with mT input neurons, dT output neurons, and T hidden layers with $N^{(2)}$ neurons each. Input $\mathbf{a}[t]^{(1)}$ is connected solely to the hidden neurons in layer t, and the hidden neurons in layer t are connected to output neurons that correspond to $\mathbf{a}[t]^{(3)}$. Furthermore, the hidden neurons in layer $t < T$ are connected to the hidden neurons in layer $t+1$. The parameters in the network are shared both between corresponding hidden neurons in distinct layers and corresponding output neurons. An analogous unfolding can be applied to any recurrent neural network architecture. Notice that the unfolded recurrent network is deep for any sequence with more than one element, which indicates that the model may be difficult to train for long sequences.

The proof of the backpropagation through time statements presented below were derived independently by the author of this text. Although the results are correct according to the literature, the details should be inspected carefully.

Consider the statement

$$\delta[t]_j^{(3)} = \frac{\partial c}{\partial a[t]_j^{(3)}} \sigma'(z[t]_j^{(3)}).$$

Because c is differentiable with respect to $a[t]_j^{(3)}$, and $z[t]_j^{(3)}$ only affects c through $a[t]_j^{(3)}$,

$$\delta[t]_j^{(3)} = \frac{\partial c}{\partial z[t]_j^{(3)}} = \frac{\partial c}{\partial a[t]_j^{(3)}} \frac{\partial a[t]_j^{(3)}}{\partial z[t]_j^{(3)}} = \frac{\partial c}{\partial a[t]_j^{(3)}} \sigma'(z[t]_j^{(3)}),$$

as we wanted to show. Furthermore, if we consider a cross-entropy cost function for sequence element classification and a sigmoid activation function,

$$\begin{aligned}
\frac{\partial c}{\partial a[t]_j^{(3)}} &= \frac{\partial}{\partial a[t]_j^{(3)}} \left[-\frac{1}{T} \sum_{t'=1}^{T} \sum_{i=1}^{d} \left[Y[t']_i \ln{(a[t']_i^{(3)})} + (1 - Y[t']_i) \ln{(1 - a[t']_i^{(3)})} \right] \right], \\
&= \frac{\partial}{\partial a[t]_j^{(3)}} \left[-\frac{1}{T} \left[Y[t]_j \ln{(a[t]_j^{(3)})} + (1 - Y[t]_j) \ln{(1 - a[t]_j^{(3)})} \right] \right], \\
&= -\frac{1}{T} \left[\frac{Y[t]_j}{a[t]_j^{(3)}} - \frac{(1 - Y[t]_j)}{(1 - a[t]_j^{(3)})} \right], \\
&= \frac{a[t]_j^{(3)} - Y[t]_j}{a[t]_j^{(3)}(1 - a[t]_j^{(3)})T}, \\
&= \frac{a[t]_j^{(3)} - Y[t]_j}{\sigma'(z[t]_j^{(3)})T},
\end{aligned}$$

which gives $\delta[t]_j^{(3)} = \frac{\partial c}{\partial a[t]_j^{(3)}} \sigma'(z[t]_j^{(3)}) = \frac{a[t]_j^{(3)} - Y[t]_j}{T}$.

In the case of a softmax output layer, it is no longer true that $z[t]_j^{(3)}$ only affects c through $a[t]_j^{(3)}$. However, it is still true that $\delta[t]_j^{(3)} = \frac{a[t]_j^{(3)} - Y[t]_j}{T}$ for a negative log-likelihood cost function c for sequence element classification. The proof is analogous to that presented in a previous section.

Consider the statement

$$\delta[t]_j^{(2)} = \sigma'(z[t]_j^{(2)}) \left[\sum_{k=1}^{N^{(3)}} w_{k,j}^{(3)} \delta[t]_k^{(3)} + \sum_{k=1}^{N^{(2)}} \omega_{k,j}^{(2)} \delta[t+1]_k^{(2)} \right],$$

which we will prove for every $t < T$. As already mentioned, we define $\boldsymbol{\delta}[T+1]^{(2)} = 0$.

Because $z[t]_j^{(2)}$ only affects c through $z[t]_1^{(3)}, \dots, z[t]_{N^{(3)}}^{(3)}$ and through $z[t+1]_1^{(2)}, \dots, z[t+1]_{N^{(2)}}^{(2)}$, the chain rule gives

$$\begin{aligned}
\delta[t]_j^{(2)} = \frac{\partial c}{\partial z[t]_j^{(2)}} &= \sum_{k=1}^{N^{(3)}} \frac{\partial c}{\partial z[t]_k^{(3)}} \frac{\partial z[t]_k^{(3)}}{\partial z[t]_j^{(2)}} + \sum_{k=1}^{N^{(2)}} \frac{\partial c}{\partial z[t+1]_k^{(2)}} \frac{\partial z[t+1]_k^{(2)}}{\partial z[t]_j^{(2)}} \\
&= \sum_{k=1}^{N^{(3)}} \delta[t]_k^{(3)} \frac{\partial z[t]_k^{(3)}}{\partial z[t]_j^{(2)}} + \sum_{k=1}^{N^{(2)}} \delta[t+1]_k^{(2)} \frac{\partial z[t+1]_k^{(2)}}{\partial z[t]_j^{(2)}}.
\end{aligned}$$

From the definition of $z[t]_k^{(3)}$,

$$\frac{\partial z[t]_k^{(3)}}{\partial z[t]_j^{(2)}} = \frac{\partial}{\partial z[t]_j^{(2)}} \left[b_k^{(3)} + \sum_{i=1}^{N^{(2)}} w_{k,i}^{(3)} a[t]_i^{(2)} \right] = w_{k,j}^{(3)} \sigma'(z[t]_j^{(2)}).$$

From the definition of $z[t+1]_k^{(2)}$,

$$\begin{aligned}\frac{\partial z[t+1]_k^{(2)}}{\partial z[t]_j^{(2)}} &= \frac{\partial}{\partial z[t]_j^{(2)}} \left[b_k^{(2)} + \sum_{i=1}^{N^{(1)}} w_{k,i}^{(2)} a[t+1]_i^{(1)} + \sum_{i=1}^{N^{(2)}} \omega_{k,i}^{(2)} a[t]_i^{(2)} \right], \\ &= \frac{\partial}{\partial z[t]_j^{(2)}} \left[\sum_{i=1}^{N^{(2)}} \omega_{k,i}^{(2)} a[t]_i^{(2)} \right], \\ &= \omega_{k,j}^{(2)} \sigma'(z[t]_j^{(2)}).\end{aligned}$$

Going back,

$$\begin{aligned}\delta[t]_j^{(2)} &= \sum_{k=1}^{N^{(3)}} \delta[t]_k^{(3)} w_{k,j}^{(3)} \sigma'(z[t]_j^{(2)}) + \sum_{k=1}^{N^{(2)}} \delta[t+1]_k^{(2)} \omega_{k,j}^{(2)} \sigma'(z[t]_j^{(2)}), \\ &= \sigma'(z[t]_j^{(2)}) \left[\sum_{k=1}^{N^{(3)}} w_{k,j}^{(3)} \delta[t]_k^{(3)} + \sum_{k=1}^{N^{(2)}} \omega_{k,j}^{(2)} \delta[t+1]_k^{(2)} \right],\end{aligned}$$

as we wanted to show.

Consider the statement $\frac{\partial c}{\partial b_j^{(l)}} = \sum_{t=1}^{T} \delta[t]_j^{(l)}$, which we prove independently for $l = 3$ and $l = 2$.

Because $b_j^{(3)}$ only affects c through $z[1]_j^{(3)}, \dots, z[T]_j^{(3)}$, the chain rule gives

$$\frac{\partial c}{\partial b_j^{(3)}} = \sum_{t=1}^{T} \frac{\partial c}{\partial z[t]_j^{(3)}} \frac{\partial z[t]_j^{(3)}}{\partial b_j^{(3)}} = \sum_{t=1}^{T} \delta[t]_j^{(3)},$$

which completes the proof for $l = 3$.

Although $b_j^{(2)}$ only affects c through $z[1]_j^{(2)}, \dots, z[T]_j^{(2)}$, it is also true that $z[t]_j^{(2)}$ may affect $z[t+1]_k^{(2)}$ for any j, k and t. By consequence, the chain rule does not apply analogously to the previous case. Consider, as an artifice, the introduction of independent biases $b[t]_j^{(2)} = b_j^{(2)}$ for each time step t. Because $b_j^{(2)}$ may only affect c through the new biases:

$$\frac{\partial c}{\partial b_j^{(2)}} = \sum_{t=1}^{T} \frac{\partial c}{b[t]_j^{(2)}} \frac{\partial b[t]_j^{(2)}}{\partial b_j^{(2)}} = \sum_{t=1}^{T} \frac{\partial c}{b[t]_j^{(2)}}.$$

Because each bias $b[t]_j^{(2)}$ only affects c through $z[t]_j^{(2)}$, we have

$$\frac{\partial c}{b[t]_j^{(2)}} = \frac{\partial c}{\partial z[t]_j^{(2)}} \frac{\partial z[t]_j^{(2)}}{\partial b_j^{(2)}} = \frac{\partial c}{\partial z[t]_j^{(2)}} = \delta[t]_j^{(2)},$$

as we wanted to show.

Consider the statement $\frac{\partial c}{\partial w_{j,k}^{(l)}} = \sum_{t=1}^{T} \delta[t]_j^{(l)} a[t]_k^{(l-1)}$, which we prove independently for $l = 2$ and $l = 3$.

Because $w_{j,k}^{(3)}$ only affects c through $z[1]_j^{(3)}, \dots, z[T]_j^{(3)}$, the chain rule gives

$$\frac{\partial c}{\partial w_{j,k}^{(3)}} = \sum_{t=1}^{T} \frac{\partial c}{\partial z[t]_j^{(3)}} \frac{\partial z[t]_j^{(3)}}{\partial w_{j,k}^{(3)}} = \sum_{t=1}^{T} \delta[t]_j^{(3)} a[t]_k^{(2)},$$

which completes the proof for $l = 3$.

Once again, $w^{(2)}_{j,k}$ only affects c through $z[1]^{(2)}_j, \dots, z[T]^{(2)}_j$, but $z[t]^{(2)}_j$ may affect $z[t+1]^{(2)}_k$ for any j, k and t, and so the chain rule does not apply analogously to the previous case. As an artifice, we introduce independent weights $w[t]^{(2)}_{j,k} = w^{(2)}_{j,k}$ for each time step t, which gives

$$\frac{\partial c}{\partial w^{(2)}_{j,k}} = \sum_{t=1}^{T} \frac{\partial c}{\partial w[t]^{(2)}_{j,k}} \frac{\partial w[t]^{(2)}_{j,k}}{\partial w^{(2)}_{j,k}} = \sum_{t=1}^{T} \frac{\partial c}{\partial w[t]^{(2)}_{j,k}}.$$

Because $w[t]^{(2)}_{j,k}$ only affects c through $z[t]^{(2)}_j$,

$$\frac{\partial c}{\partial w[t]^{(2)}_{j,k}} = \frac{\partial c}{\partial z[t]^{(2)}_j} \frac{\partial z[t]^{(2)}_j}{\partial w[t]^{(2)}_{j,k}} = \delta[t]^{(2)}_j a[t]^{(1)}_k,$$

which concludes the proof for $l = 2$.

Consider the statement

$$\frac{\partial c}{\partial \omega^{(2)}_{j,k}} = \sum_{t=1}^{T} \delta[t]^{(2)}_j a[t-1]^{(2)}_k.$$

This statement also requires the artifice of introducing independent recurrent weights $\omega[t]^{(2)}_{j,k} = \omega^{(2)}_{j,k}$ for each time step t. The chain rule gives

$$\frac{\partial c}{\partial \omega^{(2)}_{j,k}} = \sum_{t=1}^{T} \frac{\partial c}{\partial \omega[t]^{(2)}_{j,k}} \frac{\partial \omega[t]^{(2)}_{j,k}}{\partial \omega^{(2)}_{j,k}} = \sum_{t=1}^{T} \frac{\partial c}{\partial \omega[t]^{(2)}_{j,k}}.$$

Because $\omega[t]^{(2)}_{j,k}$ refers to the weight for time step t, it only affects c through $z[t]^{(2)}_j$. The the chain rule gives

$$\frac{\partial c}{\partial \omega[t]^{(2)}_{j,k}} = \frac{\partial c}{\partial z[t]^{(2)}_j} \frac{\partial z[t]^{(2)}_j}{\partial \omega[t]^{(2)}_{j,k}} = \delta[t]^{(2)}_j a[t-1]^{(2)}_k,$$

as we wanted to show.

The three last backpropagation through time statements follow easily from the definition of the cost C as the average of the cost c over every pair $(X, Y) \in \mathcal{D}$. This completes the definition of the partial derivatives of the cost function C with respect to every parameter in a single hidden layer recurrent neural network.

Stochastic gradient descent can be employed to minimize C, with the usual lack of guarantees. In practice, online gradient descent (size one mini-batches) are commonly used. The improvements suggested in previous sections, such as momentum-based gradient descent and regularization, can also be employed.

Finding partial derivatives of the cost with respect to the parameters becomes very involved for more complicated network architectures. In those cases, it is highly advisable to implement recurrent neural networks using automatic differentiation algorithms.

7 Long short-term memory networks

Long short-term memory networks compose an important class of recurrent neural networks. These networks were developed to deal with exploding and vanishing gradients, which may compromise learning in the recurrent networks presented in the previous section. Long short-term memory networks have been shown to be very effective in practice.

Consider the sequence element classification data set $\mathcal{D} = \{(X_i, Y_i) \mid i \in \{1, \dots, n\}, X_i \in (\mathbb{R}^m)^+, Y_i \in (\mathbb{R}^d)^+\}$. Furthermore, let $|X| = |Y|$ for every $(X, Y) \in \mathcal{D}$.

We will present long short-term memory networks with a single hidden layer composed of single-celled memory blocks. Many definitions will be analogous to those presented for single hidden layer recurrent neural networks and will be omitted.

Let $N^{(1)} = m$ and $N^{(3)} = d$. Let $N^{(2)}$ denote the number of single-celled memory blocks in the hidden layer. The input activation for the network at time t for $(X, Y) \in \mathcal{D}$ is defined as $X[t] = \mathbf{a}[t]^{(1)}$. Similarly to a neuron in the hidden layer of a typical recurrent neural network, memory block j also receives the vectors $\mathbf{a}[t]^{(1)}$ and $\mathbf{a}[t-1]^{(2)}$ at time step t, and outputs a scalar $a[t]^{(2)}_j$. Long short-term memory networks can be unfolded in time in an analogous manner to typical recurrent networks. However, the computations performed in a memory block are considerably more involved than those in a recurrent artificial neuron.

A single-celled memory block is composed of four *modules*: cell, input gate I, forget gate F and output gate O.

The weighted input $z[t]^{(2)}_j$ to the cell in memory block j is defined as

$$z[t]^{(2)}_j = b^{(2)}_j + \sum_{k=1}^{N^{(1)}} w^{(2)}_{j,k} a[t]^{(1)}_k + \sum_{k=1}^{N^{(2)}} \omega^{(2)}_{j,k} a[t-1]^{(2)}_k,$$

where $\mathbf{a}[0]^{(2)}$ may be zero. This is the analogous of the weighted input for neuron j in the hidden layer of a typical recurrent network.

The activation $s[t]^{(2)}_j$ of the cell in memory block j is defined as

$$s[t]^{(2)}_j = a[t]^{(2)}_{F,j} s[t-1]^{(2)}_j + a[t]^{(2)}_{I,j} g(z[t]^{(2)}_j),$$

where $\mathbf{s}[0]^{(2)}$ may be zero, and g is a differentiable activation function. The terms $a[t]^{(2)}_{F,j}$ and $a[t]^{(2)}_{I,j}$ correspond to the activations of the forget and input gates, respectively, and will be defined shortly. Because each of these two scalars is usually between 0 and 1, they control how much the previous activation of the cell and the current weighted input to the cell affect its current activation.

The weighted input $z[t]^{(2)}_{G,j}$ of a gate $G = I, F$ or O in memory block j is defined as

$$z[t]^{(2)}_{G,j} = b^{(2)}_{G,j} + \psi^{(2)}_{G,j} s[t-1]^{(2)}_j + \sum_{k=1}^{N^{(1)}} w^{(2)}_{G,j,k} a[t]^{(1)}_k + \sum_{k=1}^{N^{(2)}} \omega^{(2)}_{G,j,k} a[t-1]^{(2)}_k,$$

where $\psi_{G,j} \in \mathbb{R}$ is the so-called peephole weight to the cell activation in the previous time step.

The activation $a[t]^{(2)}_{G,j}$ of a gate G in memory block j is defined as $a[t]^{(2)}_{G,j} = f(z[t]^{(2)}_{G,j})$, where f is a differentiable activation function, which is typically the sigmoid function. Notice that each gate G in memory block j has its own parameters and behaves similarly to a typical recurrent neuron.

Finally, the output activation $a[t]^{(2)}_j$ of memory block j is defined as

$$a[t]^{(2)}_j = a[t]^{(2)}_{O,j} h(s[t]^{(2)}_j),$$

where h is a differentiable activation function. Intuitively, the activation of the output gate controls how much the current activation of the cell affects the output of the memory block.

Intuitively, a memory block can be interpreted as a parameterized circuit. By training the network, a memory block may learn when to store, output and erase its memory (cell activation), given the current input activation to the network and the previous activation of the memory blocks. Each memory block has $4(N^{(1)} + N^{(2)}) + 7$ parameters (weights and biases).

The weighted input to neuron $1 \leq j \leq N^{(3)}$ in layer 3 is defined as

$$z[t]^{(3)}_j = b^{(3)}_j + \sum_{k=1}^{N^{(2)}} w^{(3)}_{j,k} a[t]^{(2)}_k,$$

as in a feedforward neural network. The activation of neuron j in layer 3 is $a[t]^{(3)}_j = \sigma(z[t]^{(3)}_j)$.

The output of the network f on input $X = \mathbf{a}[1]^{(1)}, \dots, \mathbf{a}[T]^{(1)}$ is the sequence $\mathbf{a}[1]^{(3)}, \dots, \mathbf{a}[T]^{(3)}$. This completes the definition of a long short-term memory network with a single hidden layer of single-celled memory blocks.

The intuition behind long short-term memory networks is very similar to that for typical recurrent neural networks. The sequence X is presented to the network element by element. An ideal long short-term memory network would be capable of representing a sequence $X[1:t]$ by the activation of its memory blocks $\mathbf{a}[t]^{(2)}$ and cells $\mathbf{s}[t]^{(2)}$ to allow correct classification of $X[t+1]$.

As usual, learning consists on finding parameters that minimize a cost function $C = \frac{1}{n}\sum_{(X,Y)\in\mathcal{D}} c$. For concreteness, consider the case of a cross-entropy cost function c for sequence element classification defined as

$$c = -\frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{d}\Big[Y[t]_i \ln(a[t]^{(3)}_i) + (1 - Y[t]_i)\ln(1 - a[t]^{(3)}_i)\Big],$$

where $X = \mathbf{a}[1]^{(1)}, \dots, \mathbf{a}[T]^{(1)}$ is the input sequence and $\mathbf{a}[1]^{(3)}, \dots, \mathbf{a}[T]^{(3)}$ is the output sequence. This cost function requires that every $Y[t]$ be a standard basis vector.

We will now present the backpropagation through time statements that allow the computation of the partial derivatives of the cost function with respect to every parameter in a long short-term memory network. As usual, these partial derivatives can be used for gradient descent.

We start by defining auxiliary variables. The error $\delta[t]^{(l)}_j$ of neuron or block j in layer l at time step t is defined as

$$\delta[t]^{(l)}_j = \frac{\partial c}{\partial z[t]^{(l)}_j}.$$

The error $\delta[t]^{(2)}_{G,j}$ of gate G in layer 2 at time step t is defined as

$$\delta[t]^{(2)}_{G,j} = \frac{\partial c}{\partial z[t]^{(2)}_{G,j}}.$$

The activation error $\epsilon_a[t]^{(2)}_j$ of memory block j in layer 2 at time step t is defined as

$$\epsilon_a[t]^{(2)}_j = \frac{\partial c}{\partial a[t]^{(2)}_j}.$$

The cell activation error $\epsilon_s[t]^{(2)}_j$ of memory block j in layer 2 at time step t is defined as

$$\epsilon_s[t]^{(2)}_j = \frac{\partial c}{\partial s[t]^{(2)}_j}.$$

The following statements summarize error computation by backpropagation through time in long short-term memory networks.

$$\begin{aligned}
\delta[t]^{(3)}_j &= \frac{\partial c}{\partial a[t]^{(3)}_j} \sigma'(z[t]^{(3)}_j), && (1)\\
\delta[t]^{(2)}_{O,j} &= f'(z[t]^{(2)}_{O,j}) h(s[t]^{(2)}_j) \epsilon_a[t]^{(2)}_j, && (2)\\
\delta[t]^{(2)}_j &= a[t]^{(2)}_{I,j} g'(z[t]^{(2)}_j) \epsilon_s[t]^{(2)}_j, && (3)\\
\delta[t]^{(2)}_{F,j} &= f'(z[t]^{(2)}_{F,j}) s[t-1]^{(2)}_j \epsilon_s[t]^{(2)}_j, && (4)\\
\delta[t]^{(2)}_{I,j} &= f'(z[t]^{(2)}_{I,j}) g(z[t]^{(2)}_j) \epsilon_s[t]^{(2)}_j, && (5)\\
\epsilon_s[t]^{(2)}_j &= a[t]^{(2)}_{O,j} h'(s[t]^{(2)}_j) \epsilon_a[t]^{(2)}_j + a[t+1]^{(2)}_{F,j} \epsilon_s[t+1]^{(2)}_j + \sum_{G' \in \{I,F,O\}} \psi_{G',j} \delta[t+1]^{(2)}_{G',j}, && (6)\\
\epsilon_a[t]^{(2)}_j &= \sum_{k=1}^{N^{(3)}} w^{(3)}_{k,j} \delta[t]^{(3)}_k + \sum_{k=1}^{N^{(2)}} \omega^{(2)}_{k,j} \delta[t+1]^{(2)}_k + \sum_{G' \in \{I,F,O\}} \sum_{k=1}^{N^{(2)}} \omega^{(2)}_{G',k,j} \delta[t+1]^{(2)}_{G',k}, && (7)
\end{aligned}$$

where $\boldsymbol{\delta}[T+1]^{(2)} = \boldsymbol{\delta}[T+1]^{(2)}_G = \boldsymbol{\epsilon}_a[T+1]^{(2)} = \boldsymbol{\epsilon}_s[T+1]^{(2)} = 0$, and G is either I, F or O. Once again, every quantity can be computed easily by starting from $\boldsymbol{\delta}[T]^{(3)}$, which can only be computed after the network has observed the entire sequence.

These errors can be used to compute the partial derivative of the cost C with respect to every parameter in the network using the following statements.

$$\frac{\partial c}{\partial b^{(2)}_{G,j}} = \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j}, \tag{8}$$

$$\frac{\partial c}{\partial w^{(2)}_{G,j,k}} = \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j} a[t]^{(1)}_{k}, \tag{9}$$

$$\frac{\partial c}{\partial \omega^{(2)}_{G,j,k}} = \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j} a[t-1]^{(2)}_{k}, \tag{10}$$

$$\frac{\partial c}{\partial \psi^{(2)}_{G,j}} = \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j} s[t-1]^{(2)}_{j}, \tag{11}$$

$$\frac{\partial c}{\partial b^{(l)}_{j}} = \sum_{t=1}^{T} \delta[t]^{(l)}_{j}, \tag{12}$$

$$\frac{\partial c}{\partial w^{(l)}_{j,k}} = \sum_{t=1}^{T} \delta[t]^{(l)}_{j} a[t]^{(l-1)}_{k}, \tag{13}$$

$$\frac{\partial c}{\partial \omega^{(2)}_{j,k}} = \sum_{t=1}^{T} \delta[t]^{(2)}_{j} a[t-1]^{(2)}_{k}, \tag{14}$$

$$\frac{\partial C}{\partial p} = \frac{1}{n} \sum_{(X,Y)\in\mathcal{D}} \frac{\partial c}{\partial p}, \tag{15}$$

where p is any parameter (weight or bias) in the network, and G is either I, F or O.

We now present proofs for the aforementioned statements for backpropagation through time in long short-term memory networks.

Consider the statement $\delta[t]^{(3)}_j = \frac{\partial c}{\partial a[t]^{(3)}_j}\sigma'(z[t]^{(3)}_j)$. For concreteness, let c be the cross-entropy cost function for sequence element classification and consider a sigmoid output layer. Because c is differentiable with respect to $a[t]^{(3)}_j$, and $z[t]^{(3)}_j$ only affects c through $a[t]^{(3)}_j$,

$$\delta[t]^{(3)}_j = \frac{\partial c}{\partial z[t]^{(3)}_j} = \frac{\partial c}{\partial a[t]^{(3)}_j}\frac{\partial a[t]^{(3)}_j}{\partial z[t]^{(3)}_j} = \frac{\partial c}{\partial a[t]^{(3)}_j}\sigma'(z[t]^{(3)}_j),$$

as we wanted to show. This proof is identical to the one presented in the previous section. A sigmoid output layer also gives $\delta[t]^{(3)}_j = \frac{\partial c}{\partial a[t]^{(3)}_j}\sigma'(z[t]^{(3)}_j) = \frac{a[t]^{(3)}_j - Y[t]_j}{T}$, which was also proved in the previous section. A negative log-likelihood cost function c for sequence element classification with a softmax output layer also results in $\delta[t]^{(3)}_j = \frac{a[t]^{(3)}_j - Y[t]_j}{T}$, although the statement $\delta[t]^{(3)}_j = \frac{\partial c}{\partial a[t]^{(3)}_j}\sigma'(z[t]^{(3)}_j)$ is no longer true. The proof is straightforward given the analysis of the negative log-likelihood cost function presented in a previous section.

A dependency graph may be useful to understand the relationships between variables on which the next proofs rely.

Consider the statement $\delta[t]^{(2)}_{O,j} = f'(z[t]^{(2)}_{O,j})h(s[t]^{(2)}_j)\epsilon_a[t]^{(2)}_j$. Because $z[t]^{(2)}_{O,j}$ only affects c through $a[t]^{(2)}_j$,

$$\delta[t]^{(2)}_{O,j} = \frac{\partial c}{\partial z[t]^{(2)}_{O,j}} = \frac{\partial c}{\partial a[t]^{(2)}_j}\frac{\partial a[t]^{(2)}_j}{\partial z[t]^{(2)}_{O,j}} = \epsilon_a[t]^{(2)}_j \frac{\partial}{\partial z[t]^{(2)}_{O,j}}\left[a[t]^{(2)}_{O,j} h(s[t]^{(2)}_j)\right].$$

Because $s[t]^{(2)}_j$ is not affected by $z[t]^{(2)}_{O,j}$,

$$\delta[t]^{(2)}_{O,j} = \epsilon_a[t]^{(2)}_j h(s[t]^{(2)}_j)\frac{\partial a[t]^{(2)}_{O,j}}{\partial z[t]^{(2)}_{O,j}} = \epsilon_a[t]^{(2)}_j h(s[t]^{(2)}_j) f'(z[t]^{(2)}_{O,j}),$$

as we wanted to show.

Consider the statement $\delta[t]_j^{(2)} = a[t]_{I,j}^{(2)} g'(z[t]_j^{(2)}) \epsilon_s[t]_j^{(2)}$. Because $z[t]_j^{(2)}$ only affects c through $s[t]_j^{(2)}$,

$$\delta[t]_j^{(2)} = \frac{\partial c}{\partial z[t]_j^{(2)}} = \frac{\partial c}{\partial s[t]_j^{(2)}} \frac{\partial s[t]_j^{(2)}}{\partial z[t]_j^{(2)}} = \epsilon_s[t]_j^{(2)} \frac{\partial}{\partial z[t]_j^{(2)}} \left[a[t]_{F,j}^{(2)} s[t-1]_j^{(2)} + a[t]_{I,j}^{(2)} g(z[t]_j^{(2)}) \right].$$

Because only $g(z[t]_j^{(2)})$ is affected by $z[t]_j^{(2)}$ on the expression between brackets,

$$\delta[t]_j^{(2)} = \epsilon_s[t]_j^{(2)} a[t]_{I,j}^{(2)} g'(z[t]_j^{(2)}),$$

as we wanted to show. Notice how this error is regulated by the activation of the input gate. Intuitively, if the input gate is closed (when its activation is close to zero), weighted input parameters do not significantly affect the cost.

Consider the statement $\delta[t]_{F,j}^{(2)} = f'(z[t]_{F,j}^{(2)}) s[t-1]_j^{(2)} \epsilon_s[t]_j^{(2)}$. Because $z[t]_{F,j}^{(2)}$ only affects c through $s[t]_j^{(2)}$,

$$\delta[t]_{F,j}^{(2)} = \frac{\partial c}{\partial z[t]_{F,j}^{(2)}} = \frac{\partial c}{\partial s[t]_j^{(2)}} \frac{\partial s[t]_j^{(2)}}{\partial z[t]_{F,j}^{(2)}} = \epsilon_s[t]_j^{(2)} \frac{\partial}{\partial z[t]_{F,j}^{(2)}} \left[a[t]_{F,j}^{(2)} s[t-1]_j^{(2)} + a[t]_{I,j}^{(2)} g(z[t]_j^{(2)}) \right].$$

Because only $a[t]_{F,j}^{(2)}$ is affected by $z[t]_{F,j}^{(2)}$ on the expression between brackets,

$$\delta[t]_{F,j}^{(2)} = \epsilon_s[t]_j^{(2)} f'(z[t]_{F,j}^{(2)}) s[t-1]_j^{(2)},$$

as we wanted to show.

Consider the statement $\delta[t]_{I,j}^{(2)} = f'(z[t]_{I,j}^{(2)}) g(z[t]_j^{(2)}) \epsilon_s[t]_j^{(2)}$. Because $z[t]_{I,j}^{(2)}$ only affects c through $s[t]_j^{(2)}$,

$$\delta[t]_{I,j}^{(2)} = \frac{\partial c}{\partial z[t]_{I,j}^{(2)}} = \frac{\partial c}{\partial s[t]_j^{(2)}} \frac{\partial s[t]_j^{(2)}}{\partial z[t]_{I,j}^{(2)}} = \epsilon_s[t]_j^{(2)} \frac{\partial}{\partial z[t]_{I,j}^{(2)}} \left[a[t]_{F,j}^{(2)} s[t-1]_j^{(2)} + a[t]_{I,j}^{(2)} g(z[t]_j^{(2)}) \right].$$

Because only $a[t]_{I,j}^{(2)}$ is affected by $z[t]_{I,j}^{(2)}$ on the expression between brackets,

$$\delta[t]_{I,j}^{(2)} = \epsilon_s[t]_j^{(2)} f'(z[t]_{I,j}^{(2)}) g(z[t]_j^{(2)}),$$

as we wanted to show.

Consider the statement

$$\epsilon_s[t]_j^{(2)} = a[t]_{O,j}^{(2)} h'(s[t]_j^{(2)}) \epsilon_a[t]_j^{(2)} + a[t+1]_{F,j}^{(2)} \epsilon_s[t+1]_j^{(2)} + \sum_{G' \in \{I,F,O\}} \psi_{G',j} \delta[t+1]_{G',j}^{(2)}.$$

Firstly, notice that $s[t]_j^{(2)}$ only affects c through $a[t]_j^{(2)}$, $s[t+1]_j^{(2)}$ and $z[t+1]_{G',j}^{(2)}$, for every $G' \in \{I, F, O\}$. However, $a[t]_j^{(2)}$ affects $z[t+1]_{G',j}^{(2)}$, and $z[t+1]_{I,j}^{(2)}$ and $z[t+1]_{F,j}^{(2)}$ also affect $s[t+1]_j^{(2)}$. Therefore, the chain rule does not apply. Consider, as an artifact, a new parameter $p[t]_j^{(2)} = s[t-1]_j^{(2)}$ and redefine $z[t]_{G,j}^{(2)}$ as

$$z[t]_{G,j}^{(2)} = b_{G,j}^{(2)} + \psi_{G,j}^{(2)} p[t]_j^{(2)} + \sum_{k=1}^{N^{(1)}} w_{G,j,k}^{(2)} a[t]_k^{(1)} + \sum_{k=1}^{N^{(2)}} \omega_{G,j,k}^{(2)} a[t-1]_k^{(2)},$$

for all $G \in \{I, F, O\}$.

Also, introduce a new parameter $p'[t]_j^{(2)} = s[t-1]_j^{(2)}$ and redefine $s[t]_j^{(2)}$ as

$$s[t]_j^{(2)} = a[t]_{F,j}^{(2)} p'[t]_j^{(2)} + a[t]_{I,j}^{(2)} g(z[t]_j^{(2)}).$$

It is easy to see that these new parameters do not affect forward propagation or the backpropagation statements proved so far.

Using these artifacts, $s[t]_j^{(2)}$ only affects c through $a[t]_j^{(2)}$, $p[t+1]_j^{(2)}$ and $p'[t+1]_j^{(2)}$. Therefore,

$$\epsilon_s[t]_j^{(2)} = \frac{\partial c}{\partial s[t]_j^{(2)}} = \frac{\partial c}{\partial a[t]_j^{(2)}} \frac{\partial a[t]_j^{(2)}}{\partial s[t]_j^{(2)}} + \frac{\partial c}{\partial p[t+1]_j^{(2)}} \frac{\partial p[t+1]_j^{(2)}}{\partial s[t]_j^{(2)}} + \frac{\partial c}{\partial p'[t+1]_j^{(2)}} \frac{\partial p'[t+1]_j^{(2)}}{\partial s[t]_j^{(2)}}.$$

Consider the first term for $\epsilon_s[t]_j^{(2)}$. By definition,

$$\frac{\partial c}{\partial a[t]_j^{(2)}} \frac{\partial a[t]_j^{(2)}}{\partial s[t]_j^{(2)}} = \epsilon_a[t]_j^{(2)} \frac{\partial}{\partial s[t]_j^{(2)}} \left[a[t]_{O,j}^{(2)} h(s[t]_j^{(2)}) \right].$$

Because $a[t]_{O,j}^{(2)}$ is not affected by $s[t]_j^{(2)}$,

$$\frac{\partial c}{\partial a[t]_j^{(2)}} \frac{\partial a[t]_j^{(2)}}{\partial s[t]_j^{(2)}} = \epsilon_a[t]_j^{(2)} a[t]_{O,j}^{(2)} h'(s[t]_j^{(2)}).$$

Consider the second term for $\epsilon_s[t]_j^{(2)}$. By definition,

$$\frac{\partial c}{\partial p[t+1]_j^{(2)}} \frac{\partial p[t+1]_j^{(2)}}{\partial s[t]_j^{(2)}} = \frac{\partial c}{\partial p[t+1]_j^{(2)}} \frac{\partial s[t]_j^{(2)}}{\partial s[t]_j^{(2)}} = \frac{\partial c}{\partial p[t+1]_j^{(2)}},$$

using the artifact variable $p[t+1]_j^{(2)} = s[t]_j^{(2)}$. Because $p[t+1]_j^{(2)}$ only affects c through $z[t+1]_{G',j}^{(2)}$ for every $G' \in \{I, F, O\}$,

$$\frac{\partial c}{\partial p[t+1]_j^{(2)}} = \sum_{G' \in \{I,F,O\}} \frac{\partial c}{\partial z[t+1]_{G',j}^{(2)}} \frac{\partial z[t+1]_{G',j}^{(2)}}{\partial p[t+1]_j^{(2)}} = \sum_{G' \in \{I,F,O\}} \delta[t+1]_{G',j}^{(2)} \frac{\partial z[t+1]_{G',j}^{(2)}}{\partial p[t+1]_j^{(2)}}.$$

Finally,

$$\frac{\partial z[t+1]_{G',j}^{(2)}}{\partial p[t+1]_j^{(2)}} = \frac{\partial}{\partial p[t+1]_j^{(2)}} \left[b_{G',j}^{(2)} + \psi_{G',j}^{(2)} p[t+1]_j^{(2)} + \sum_{k=1}^{N^{(1)}} w_{G',j,k}^{(2)} a[t+1]_k^{(1)} + \sum_{k=1}^{N^{(2)}} \omega_{G',j,k}^{(2)} a[t]_k^{(2)} \right] = \psi_{G',j}^{(2)}.$$

Going back to the term of interest,

$$\frac{\partial c}{\partial p[t+1]_j^{(2)}} \frac{\partial p[t+1]_j^{(2)}}{\partial s[t]_j^{(2)}} = \sum_{G' \in \{I,F,O\}} \delta[t+1]_{G',j}^{(2)} \psi_{G',j}^{(2)}.$$

Consider the third term for $\epsilon_s[t]_j^{(2)}$. By definition,

$$\frac{\partial c}{\partial p'[t+1]_j^{(2)}} \frac{\partial p'[t+1]_j^{(2)}}{\partial s[t]_j^{(2)}} = \frac{\partial c}{\partial p'[t+1]_j^{(2)}} \frac{\partial s[t]_j^{(2)}}{\partial s[t]_j^{(2)}} = \frac{\partial c}{\partial p'[t+1]_j^{(2)}},$$

using the artifact variable $p'[t+1]_j^{(2)} = s[t]_j^{(2)}$. Because $p'[t+1]_j^{(2)}$ only affects c through $s[t+1]_j^{(2)}$,

$$\frac{\partial c}{\partial p'[t+1]_j^{(2)}} = \frac{\partial c}{\partial s[t+1]_j^{(2)}} \frac{\partial s[t+1]_j^{(2)}}{\partial p'[t+1]_j^{(2)}} = \epsilon_s[t+1]_j^{(2)} \frac{\partial}{\partial p'[t+1]_j^{(2)}} \left[a[t+1]_{F,j}^{(2)} p'[t+1]_j^{(2)} + a[t+1]_{I,j}^{(2)} g(z[t+1]_j^{(2)}) \right].$$

Because $p[t+1]_j^{(2)}$ does not affect either $a[t+1]_{F,j}^{(2)}$, $a[t+1]_{I,j}^{(2)}$ or $g(z[t+1]_j^{(2)})$,

$$\frac{\partial c}{\partial p[t+1]_j^{(2)}} = \epsilon_s[t+1]_j^{(2)} a[t+1]_{F,j}^{(2)}.$$

Thus,

$$\epsilon_s[t]_j^{(2)} = \epsilon_a[t]_j^{(2)} a[t]_{O,j}^{(2)} h'(s[t]_j^{(2)}) + \epsilon_s[t+1]_j^{(2)} a[t+1]_{F,j}^{(2)} + \sum_{G' \in \{I,F,O\}} \delta[t+1]_{G',j}^{(2)} \psi_{G',j}^{(2)},$$

as we wanted to show. Notice that a closed output gate implies that the cell activation error is not affected by the current activation error. Also, a closed forget gate implies that the cell activation error is not affected by the next cell activation error.

Consider the statement

$$\epsilon_a[t]^{(2)}_j = \sum_{k=1}^{N^{(3)}} w^{(3)}_{k,j}\delta[t]^{(3)}_k + \sum_{k=1}^{N^{(2)}} \omega^{(2)}_{k,j}\delta[t+1]^{(2)}_k + \sum_{G' \in \{I,F,O\}} \sum_{k=1}^{N^{(2)}} \omega^{(2)}_{G',k,j}\delta[t+1]^{(2)}_{G',k}.$$

Because $a[t]^{(2)}_j$ only affects c through $z[t]^{(3)}_k$ (for $1 \leq k \leq N^{(3)}$), through $z[t+1]^{(2)}_k$ (for $1 \leq k \leq N^{(2)}$) and through $z[t+1]^{(2)}_{G',k}$ (for $G' \in \{I, F, O\}$ and $1 \leq k \leq N^{(2)}$),

$$\begin{aligned}\epsilon_a[t]^{(2)}_j = \frac{\partial c}{\partial a[t]^{(2)}_j} &= \sum_{k=1}^{N^{(3)}} \frac{\partial c}{\partial z[t]^{(3)}_k}\frac{\partial z[t]^{(3)}_k}{\partial a[t]^{(2)}_j} + \sum_{k=1}^{N^{(2)}} \frac{\partial c}{\partial z[t+1]^{(2)}_k}\frac{\partial z[t+1]^{(2)}_k}{\partial a[t]^{(2)}_j} + \sum_{G' \in \{I,F,O\}} \sum_{k=1}^{N^{(2)}} \frac{\partial c}{\partial z[t+1]^{(2)}_{G',k}}\frac{\partial z[t+1]^{(2)}_{G',k}}{\partial a[t]^{(2)}_j}\\
&= \sum_{k=1}^{N^{(3)}} \delta[t]^{(3)}_k \frac{\partial z[t]^{(3)}_k}{\partial a[t]^{(2)}_j} + \sum_{k=1}^{N^{(2)}} \delta[t+1]^{(2)}_k \frac{\partial z[t+1]^{(2)}_k}{\partial a[t]^{(2)}_j} + \sum_{G' \in \{I,F,O\}} \sum_{k=1}^{N^{(2)}} \delta[t+1]^{(2)}_{G',k} \frac{\partial z[t+1]^{(2)}_{G',k}}{\partial a[t]^{(2)}_j}\\
&= \sum_{k=1}^{N^{(3)}} \delta[t]^{(3)}_k w^{(3)}_{k,j} + \sum_{k=1}^{N^{(2)}} \delta[t+1]^{(2)}_k \omega^{(2)}_{k,j} + \sum_{G' \in \{I,F,O\}} \sum_{k=1}^{N^{(2)}} \delta[t+1]^{(2)}_{G',k}\omega^{(2)}_{G',k,j},\end{aligned}$$

as we wanted to show.

We now present the proofs for the statements regarding the partial derivatives of c with respect to every parameter in the long short-term memory network.

Consider the parameters of a gate G in memory block j: $b^{(2)}_{G,j}, w^{(2)}_{G,j,k}, \omega^{(2)}_{G,j,i}$ and $\psi^{(2)}_{G,j}$ (for $1 \leq k \leq N^{(1)}$ and $1 \leq i \leq N^{(2)}$). Let p denote any of these parameters. It is clear that p only affects c through $z[1]^{(2)}_{G,j}, \dots, z[T]^{(2)}_{G,j}$. However, $z[t]^{(2)}_{G,j}$ may affect $z[t+1]^{(2)}_{G,j}$, and thus the chain rule does not apply. As an artifact, consider introducing an independent parameter $p[t] = p$ for each time step t. By the chain rule,

$$\frac{\partial c}{\partial p} = \sum_{i=1}^{T} \frac{\partial c}{\partial p[t]}\frac{\partial p[t]}{\partial p} = \sum_{i=1}^{T} \frac{\partial c}{\partial p[t]}.$$

Because $p[t]$ only affects c through $z[t]^{(2)}_{G,j}$,

$$\frac{\partial c}{\partial p[t]} = \frac{\partial c}{\partial z[t]^{(2)}_{G,j}}\frac{\partial z[t]^{(2)}_{G,j}}{\partial p[t]} = \delta[t]^{(2)}_{G,j}\frac{\partial z[t]^{(2)}_{G,j}}{\partial p[t]}.$$

When developed for each parameter of a gate G, this gives

$$\begin{aligned}\frac{\partial c}{\partial b^{(2)}_{G,j}} &= \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j},\\
\frac{\partial c}{\partial w^{(2)}_{G,j,k}} &= \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j} a[t]^{(1)}_k,\\
\frac{\partial c}{\partial \omega^{(2)}_{G,j,k}} &= \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j} a[t-1]^{(2)}_k,\\
\frac{\partial c}{\partial \psi^{(2)}_{G,j}} &= \sum_{t=1}^{T} \delta[t]^{(2)}_{G,j} s[t-1]^{(2)}_j.\end{aligned}$$

An analogous argument gives

$$\frac{\partial c}{\partial \omega_{j,k}^{(2)}} = \sum_{t=1}^{T} \delta[t]_j^{(2)} a[t-1]_k^{(2)},$$

$$\frac{\partial c}{\partial b_j^{(l)}} = \sum_{t=1}^{T} \delta[t]_j^{(l)},$$

$$\frac{\partial c}{\partial w_{j,k}^{(l)}} = \sum_{t=1}^{T} \delta[t]_j^{(l)} a[t]_k^{(l-1)},$$

for $l = 2$ and $l = 3$. This concludes the proofs for the statements regarding the partial derivatives of c with respect to every parameter in the long short-term memory network.

Because $C = \frac{1}{n} \sum_{(X,Y) \in \mathcal{D}} c$, it is easy to see that $\frac{\partial C}{\partial p} = \frac{1}{n} \sum_{(X,Y) \in \mathcal{D}} \frac{\partial c}{\partial p}$ for any parameter p in the network.

Each memory block in a long short-term memory network may also have more than one cell. In that case, each cell has its independent weighted input, but all cells in a memory block share the same input, output and forget gates. These gates control how the output activation *vector* of the block is affected by each independent cell activation.

8 Restricted Boltzmann machines

Restricted Boltzmann machines are probabilistic graphical models (more specifically, Markov networks) that can be used to model a joint probability distribution over a set of binary random variables. For background on probabilistic graphical models, see the corresponding notes.

A Gibbs distribution P over the set of random variables $\mathcal{X} = \{X_1, \dots, X_m\}$ parameterized by a set of non-negative factors $\Phi = \{\phi_1(D_1), \dots, \phi_K(D_K)\}$ can be written as

$$P(\mathcal{X}) = \frac{1}{Z} \prod_{i=1}^{K} \phi_i(D_i),$$

where

$$Z = \sum_{\mathcal{X}} \prod_{i=1}^{K} \phi_i(D_i)$$

is a constant.

Intuitively, a factor $\phi_i(D_i)$ represents the *joint affinity* of an assignment to the variables $D_i \subseteq \mathcal{X}$.

If all factors $\phi_i(D_i)$ are positive, it is possible to define factors $\epsilon_i(D_i) = -\ln \phi_i(D_i)$ such that

$$P(\mathcal{X}) = \frac{1}{Z} \prod_{i=1}^{K} \phi_i(D_i) = \frac{1}{Z} \prod_{i=1}^{K} e^{-\epsilon_i(D_i)} = \frac{1}{Z} e^{-\sum_{i=1}^{K} \epsilon_i(D_i)},$$

and

$$Z = \sum_{\mathcal{X}} e^{-\sum_{i=1}^{K} \epsilon_i(D_i)}.$$

By defining the energy factor $E(\mathcal{X}) = \sum_{i=1}^{K} \epsilon_i(D_i)$, P can be conveniently written as

$$P(\mathcal{X}) = \frac{1}{Z} e^{-E(\mathcal{X})}.$$

In this case, lower energy assignments to $\mathcal{X}$ correspond to higher probabilities, and the probabilistic model is said to be energy-based.

A restricted Boltzmann machine is an energy-based probabilistic model over a set of variables $\mathcal{X}$. This set is partitioned into a set of visible variables $\mathcal{V} \subseteq \mathcal{X}$, and a set of hidden variables $\mathcal{H} \subseteq \mathcal{X}$. We let $\mathbf{v} \in \text{Val}(\mathcal{V})$ denote an assignment to the variables in $\mathcal{V}$, and use an analogous notation for assignments to $\mathcal{H}$. We denote a joint assignment to $\mathcal{X}$ as $(\mathbf{v}, \mathbf{h})$ or $\mathbf{x} \in \text{Val}(\mathcal{X})$.

In a restricted Boltzmann machine, the energy $E(\mathbf{v}, \mathbf{h})$ of an assignment $(\mathbf{v}, \mathbf{h}) \in \text{Val}(\mathcal{X})$ is given by

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i=1}^{D}\sum_{j=1}^{d} w_{i,j} h_i v_j - \sum_{i=1}^{d} b_i v_i - \sum_{i=1}^{D} c_i h_i,$$

where D is the number of hidden variables, $w_{i,j} \in \mathbb{R}$ is a weight, and $b_i, c_i \in \mathbb{R}$ are biases. The relationship between these parameters and the weights and biases in feedforward neural networks will become clear as we present the properties of this model. Notice that the energy is decreased when $h_i = v_j = 1$ and $w_{i,j}$ is increased.

The probability distribution defined by a restricted Boltzmann machine can be written as

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v}, \mathbf{h})},$$

where

$$Z = \sum_{\mathbf{v}'}\sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}')}.$$

Alternatively, to make clear the dependency on the assignment $\boldsymbol{\theta}$ to the parameters (weights and biases) of the model, we can write P as

$$P(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}.$$

Clearly, the energy $E(\mathbf{v}, \mathbf{h})$ of the assignment $(\mathbf{v}, \mathbf{h})$ can also be written as the sum of the following terms:

$$\begin{aligned} \epsilon_{i,j}(h_i, v_j) &= -w_{i,j} h_i v_j, \\ \epsilon_i^{(h)}(h_i) &= -c_i h_i, \\ \epsilon_j^{(v)}(v_j) &= -b_j v_j, \end{aligned}$$

for $1 \leq i \leq D$ and $1 \leq j \leq d$. By associating each of these factors $\epsilon(D)$ with a factor $\phi(D) = e^{-\epsilon(D)}$, the distribution defined by the restricted Boltzmann machine can also be written as

$$P(\mathcal{X}) = \frac{1}{Z} \prod_{k=1}^{K} \phi_k(D_k).$$

Notice that there is single factor involving each pair (H_i, V_j), and a single factor for each variable $X_i \in \mathcal{X}$. Thus, the Gibbs distribution factorizes over a complete bipartite graph (over $\mathcal{V}$ and $\mathcal{H}$). By consequence, the graphical model encodes the following conditional independency statements: $H_i \perp\!\!\!\perp \mathcal{H}_{-i} \mid \mathcal{V}$, and $V_i \perp\!\!\!\perp \mathcal{V}_{-i} \mid \mathcal{H}$, where $\mathcal{X}_{-i} = \mathcal{X} - \{X_i\}$.

Consider a data set $\mathcal{D} = \{\mathbf{v}_i \mid i \in \{1, \ldots, n\}, \mathbf{v}_i \in \{0, 1\}^d\}$ composed of independent, identically distributed sample elements from an unknown joint probability distribution P^* over a set $\mathcal{V}$ composed of d binary random variables. We are interested in the task of learning a probability distribution P that *generalizes* well from the data in $\mathcal{D}$.

As an example, the empirical joint probability distribution defined by $\mathcal{D}$ describes the data set perfectly, but is not expected to generalize well if d is large and n (sample size) is comparatively small. There is a large number of possible joint assignments to $\mathcal{V}$ (namely, 2^d), and each assignment induces a parameter in the empirical distribution. Independency assumptions about the variables in $\mathcal{V}$ decrease the number of parameters required to define a probability distribution over $\mathcal{V}$.

A restricted Boltzmann machine can be used to model a probability distribution over the set of visible variables $\mathcal{V}$ by marginalizing over its set of hidden variables $\mathcal{H}$:

$$P(\mathbf{v}) = \sum_{\mathbf{h}} P(\mathbf{v}, \mathbf{h}).$$

The hidden variables are introduced in the model to enable the representation of an arbitrary probability distribution over $\mathcal{V}$ using at most pairwise factors (given enough hidden variables). Notice that each pairwise factor in a restricted Boltzmann machine requires a single parameter.

The goal of learning the parameters $\boldsymbol{\theta}$ for a probability distribution P defined by a restricted Boltzmann machine can be reformulated as finding parameters $\boldsymbol{\theta}^*$ with maximum likelihood for the data set $\mathcal{D}$.

The likelihood $L(\boldsymbol{\theta} : \mathcal{D})$ of the parameters $\boldsymbol{\theta}$ for the data set $\mathcal{D}$ is defined as

$$L(\boldsymbol{\theta} : \mathcal{D}) = P(\mathbf{v}_1, \dots, \mathbf{v}_n : \boldsymbol{\theta}) = \prod_{i=1}^{n} P(\mathbf{v}_i : \boldsymbol{\theta}).$$

The last equality follows from our assumption of independent, identically distributed sample elements. Maximizing the likelihood corresponds to maximizing the log-likelihood $\ell(\boldsymbol{\theta} : \mathcal{D})$:

$$\ell(\boldsymbol{\theta} : \mathcal{D}) = \ln L(\boldsymbol{\theta} : \mathcal{D}) = \ln \left[\prod_{i=1}^{n} P(\mathbf{v}_i : \boldsymbol{\theta}) \right] = \sum_{i=1}^{n} \ln P(\mathbf{v}_i : \boldsymbol{\theta})$$

For convenience, we let $\ell(\boldsymbol{\theta} : \mathbf{v}) = \ln P(\mathbf{v} : \boldsymbol{\theta})$ denote the log-likelihood of the parameters $\boldsymbol{\theta}$ for a single observation $\mathbf{v} \in \text{Val}(\mathcal{V})$. Therefore, $\ell(\boldsymbol{\theta} : \mathcal{D}) = \sum_{i=1}^{n} \ell(\boldsymbol{\theta} : \mathbf{v}_i)$.

In a restricted Boltzmann machine, the log-likelihood of the parameters $\boldsymbol{\theta}$ for a single observation can be written as

$$\ell(\boldsymbol{\theta} : \mathbf{v}) = \ln P(\mathbf{v} : \boldsymbol{\theta}) = \ln \left[\sum_{\mathbf{h}} P(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta}) \right] = \ln \left[\sum_{\mathbf{h}} \frac{e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}}{Z(\boldsymbol{\theta})} \right] = \ln \left[\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \right] - \ln Z(\boldsymbol{\theta}).$$

It is not possible to find maximum log-likelihood parameters $\boldsymbol{\theta}^*$ analytically. As usual, maximization will be attempted by gradient ascent. This requires the partial derivative of $\ell(\boldsymbol{\theta} : \mathcal{D})$ with respect to every parameter θ in the network:

$$\frac{\partial \ell(\boldsymbol{\theta} : \mathcal{D})}{\partial \theta} = \frac{\partial}{\partial \theta} \left[\sum_{i=1}^{n} \ell(\boldsymbol{\theta} : \mathbf{v}_i) \right] = \sum_{i=1}^{n} \frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v}_i)}{\partial \theta}.$$

Therefore, in order to compute $\frac{\partial \ell(\boldsymbol{\theta} : \mathcal{D})}{\partial \theta}$, it is sufficient to know the partial derivative of $\ell(\boldsymbol{\theta} : \mathbf{v}_i)$ with respect to θ for each observation $\mathbf{v}_i$ in $\mathcal{D}$. We will now prove that the desired derivative is given by

$$\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial \theta} = -\sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v} : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}{\partial \theta} + \sum_{\mathbf{v}'} P(\mathbf{v}' : \boldsymbol{\theta}) \left[\sum_{\mathbf{h}'} P(\mathbf{h}' \mid \mathbf{v}' : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})}{\partial \theta} \right].$$

From what we already showed,

$$\ell(\boldsymbol{\theta} : \mathbf{v}) = \ln \left[\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \right] - \ln Z(\boldsymbol{\theta}) = \ln \left[\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \right] - \ln \left[\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})} \right].$$

Therefore,

$$\begin{aligned}
\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial \theta} &= \frac{\partial}{\partial \theta} \left[\ln \left[\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \right] \right] - \frac{\partial}{\partial \theta} \left[\ln \left[\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})} \right] \right] \\
&= \frac{1}{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}} \frac{\partial}{\partial \theta} \left[\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \right] - \frac{1}{\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})}} \frac{\partial}{\partial \theta} \left[\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})} \right] \\
&= \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \frac{\partial}{\partial \theta} \left[-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta}) \right]}{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}} - \frac{\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})} \frac{\partial}{\partial \theta} \left[-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta}) \right]}{\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})}} \\
&= -\frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \frac{\partial}{\partial \theta} \left[E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta}) \right]}{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}} + \frac{\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})} \frac{\partial}{\partial \theta} \left[E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta}) \right]}{Z(\boldsymbol{\theta})}.
\end{aligned}$$

As an artifice, we divide both the numerator and denominator of the first term by $Z(\boldsymbol{\theta})$:

$$\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial \theta} = -\frac{\frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})} \frac{\partial}{\partial \theta} \left[E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta}) \right]}{Z(\boldsymbol{\theta})}}{\frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}}{Z(\boldsymbol{\theta})}} + \frac{\sum_{\mathbf{v}'} \sum_{\mathbf{h}'} e^{-E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})} \frac{\partial}{\partial \theta} \left[E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta}) \right]}{Z(\boldsymbol{\theta})}.$$

From the definitions of $P(\mathbf{v})$ and $P(\mathbf{v}, \mathbf{h})$:

$$
\begin{aligned}
\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial \theta} &= -\frac{\sum_{\mathbf{h}} P(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}{\partial \theta}}{P(\mathbf{v} : \boldsymbol{\theta})} + \sum_{\mathbf{v}'} \sum_{\mathbf{h}'} P(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})}{\partial \theta} \\
&= -\sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v} : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}{\partial \theta} + \sum_{\mathbf{v}'} \sum_{\mathbf{h}'} P(\mathbf{v}' : \boldsymbol{\theta}) P(\mathbf{h}' \mid \mathbf{v}' : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})}{\partial \theta} \\
&= -\sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v} : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}, \mathbf{h} : \boldsymbol{\theta})}{\partial \theta} + \sum_{\mathbf{v}'} P(\mathbf{v}' : \boldsymbol{\theta}) \sum_{\mathbf{h}'} P(\mathbf{h}' \mid \mathbf{v}' : \boldsymbol{\theta}) \frac{\partial E(\mathbf{v}', \mathbf{h}' : \boldsymbol{\theta})}{\partial \theta},
\end{aligned}
$$

as we wanted to show.

The conditional probability $P(\mathbf{h}' \mid \mathbf{v}')$ for every $\mathbf{h}'$ and $\mathbf{v}'$ is clearly important to compute $\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial \theta}$. Two remarkable properties make the former very easy to compute, as we will show.

Firstly, because $H_i \perp\!\!\!\perp \mathcal{H}_{-i} \mid \mathcal{V}$ for every i, $P(\mathbf{h} \mid \mathbf{v}) = \prod_{i=1}^{D} P(h_i \mid \mathbf{v})$, for every $\mathbf{h}$ and $\mathbf{v}$.

Secondly, we will show that $P(H_i = 1 \mid \mathbf{v}) = \sigma(c_i + \sum_{k=1}^{d} w_{i,k} v_k)$ for every i. In other words, the probability of H_i being active given an input $\mathbf{v}$ is precisely the output of a sigmoid neuron that receives $\mathbf{v}$ as input (and uses the weights and biases corresponding to H_i). The proof follows.

$$
\begin{aligned}
P(H_i = 1 \mid \mathbf{v}) = P(H_i = 1 \mid \mathbf{v}, \mathbf{h}_{-i}) = \frac{P(H_i = 1, \mathbf{v}, \mathbf{h}_{-i})}{P(\mathbf{v}, \mathbf{h}_{-i})} &= \frac{P(H_i = 1, \mathbf{v}, \mathbf{h}_{-i})}{P(H_i = 1, \mathbf{v}, \mathbf{h}_{-i}) + P(H_i = 0, \mathbf{v}, \mathbf{h}_{-i})} \\
= \frac{1}{1 + \frac{P(H_i = 0, \mathbf{v}, \mathbf{h}_{-i})}{P(H_i = 1, \mathbf{v}, \mathbf{h}_{-i})}} = \frac{1}{1 + \frac{\frac{e^{-E(H_i = 0, \mathbf{v}, \mathbf{h}_{-i})}}{Z}}{\frac{e^{-E(H_i = 1, \mathbf{v}, \mathbf{h}_{-i})}}{Z}}} &= \frac{1}{1 + \frac{e^{-E(H_i = 0, \mathbf{v}, \mathbf{h}_{-i})}}{e^{-E(H_i = 1, \mathbf{v}, \mathbf{h}_{-i})}}} \\
= \frac{1}{1 + \frac{\exp(\sum_{j=1, j \neq i}^{D} \sum_{k=1}^{d} w_{j,k} h_j v_k + \sum_{j=1}^{d} b_j v_j + \sum_{j=1, j \neq i}^{D} c_j h_j)}{\exp(\sum_{j=1, j \neq i}^{D} \sum_{k=1}^{d} w_{j,k} h_j v_k + \sum_{j=1}^{d} b_j v_j + \sum_{j=1, j \neq i}^{D} c_j h_j + c_i + \sum_{k=1}^{d} w_{i,k} v_k)}} \\
= \frac{1}{1 + \frac{\exp(\sum_{j=1, j \neq i}^{D} \sum_{k=1}^{d} w_{j,k} h_j v_k + \sum_{j=1}^{d} b_j v_j + \sum_{j=1, j \neq i}^{D} c_j h_j)}{\exp(\sum_{j=1, j \neq i}^{D} \sum_{k=1}^{d} w_{j,k} h_j v_k + \sum_{j=1}^{d} b_j v_j + \sum_{j=1, j \neq i}^{D} c_j h_j) \exp(c_i + \sum_{k=1}^{d} w_{i,k} v_k)}} \\
= \frac{1}{1 + \frac{1}{\exp(c_i + \sum_{k=1}^{d} w_{i,k} v_k)}} = \frac{1}{1 + e^{-c_i - \sum_{k=1}^{d} w_{i,k} v_k)}} = \sigma\Big(c_i + \sum_{k=1}^{d} w_{i,k} v_k\Big),
\end{aligned}
$$

as we wanted to show. Notice that the proof depends on H_i being a binary random variable, and on $P(H_i = 1, \mathbf{v}, \mathbf{h}_{-i})$ being always non-zero, which is true in an energy-based model.

An analogous proof shows that $P(V_i = 1 \mid \mathbf{h}) = \sigma(b_i + \sum_{k=1}^{D} w_{k,i} h_k)$.

Finally, we will show how these results can be used to arrive at $\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial w_{i,j}}$, $\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial b_i}$ and $\frac{\partial \ell(\boldsymbol{\theta} : \mathbf{v})}{\partial c_i}$ for all valid i and j.

Firstly, from the definition of $E(\mathbf{v}, \mathbf{h})$,

$$
\begin{aligned}
\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial w_{i,j}} &= -h_i v_j \\
\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial b_i} &= -v_i \\
\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial c_i} &= -h_i.
\end{aligned}
$$

From the general formula for $\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial \theta}$,

$$\begin{aligned}
\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial w_{i,j}} &= \sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v}) h_i v_j - \sum_{\mathbf{v}'} P(\mathbf{v}') \sum_{\mathbf{h}'} P(\mathbf{h}' \mid \mathbf{v}') h'_i v'_j \\
&= v_j \sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v}) h_i - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j \sum_{\mathbf{h}'} P(\mathbf{h}' \mid \mathbf{v}') h'_i \\
&= v_j \sum_{\mathbf{h}} P(\mathbf{h}_{-i} \mid \mathbf{v}) P(h_i \mid \mathbf{v}) h_i - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j \sum_{\mathbf{h}'} P(\mathbf{h}'_{-i} \mid \mathbf{v}') P(h'_i \mid \mathbf{v}') h'_i \\
&= v_j \sum_{h_i} \sum_{\mathbf{h}_{-i}} P(\mathbf{h}_{-i} \mid \mathbf{v}) P(h_i \mid \mathbf{v}) h_i - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j \sum_{h'_i} \sum_{\mathbf{h}'_{-i}} P(\mathbf{h}'_{-i} \mid \mathbf{v}') P(h'_i \mid \mathbf{v}') h'_i \\
&= v_j \sum_{h_i} P(h_i \mid \mathbf{v}) h_i \sum_{\mathbf{h}_{-i}} P(\mathbf{h}_{-i} \mid \mathbf{v}) - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j \sum_{h'_i} P(h'_i \mid \mathbf{v}') h'_i \sum_{\mathbf{h}'_{-i}} P(\mathbf{h}'_{-i} \mid \mathbf{v}') \\
&= v_j \sum_{h_i} P(h_i \mid \mathbf{v}) h_i - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j \sum_{h'_i} P(h'_i \mid \mathbf{v}') h'_i \\
&= v_j \big[P(H_i = 0 \mid \mathbf{v}) \times 0 + P(H_i = 1 \mid \mathbf{v}) \times 1 \big] - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j \big[P(H_i = 0 \mid \mathbf{v}') \times 0 + P(H_i = 1 \mid \mathbf{v}') \times 1 \big] \\
&= v_j P(H_i = 1 \mid \mathbf{v}) - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j P(H_i = 1 \mid \mathbf{v}') \\
&= v_j \sigma(c_i + \sum_{k=1}^{d} w_{i,k} v_k) - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_j \sigma(c_i + \sum_{k=1}^{d} w_{i,k} v'_k).
\end{aligned}$$

Notice that the first term on the right side is very easy to compute, and that the second term is the same for any $\mathbf{v} \in \mathcal{D}$.

From the general formula for $\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial \theta}$,

$$\begin{aligned}
\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial c_i} &= \sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v}) h_i - \sum_{\mathbf{v}'} P(\mathbf{v}') \sum_{\mathbf{h}'} P(\mathbf{h}' \mid \mathbf{v}') h'_i \\
&= \sum_{h_i} \sum_{\mathbf{h}_{-i}} P(\mathbf{h}_{-i} \mid \mathbf{v}) P(h_i \mid \mathbf{v}) h_i - \sum_{\mathbf{v}'} P(\mathbf{v}') \sum_{h'_i} \sum_{\mathbf{h}'_{-i}} P(\mathbf{h}'_{-i} \mid \mathbf{v}') P(h'_i \mid \mathbf{v}) h'_i \\
&= \sum_{h_i} P(h_i \mid \mathbf{v}) h_i \sum_{\mathbf{h}_{-i}} P(\mathbf{h}_{-i} \mid \mathbf{v}) - \sum_{\mathbf{v}'} P(\mathbf{v}') \sum_{h'_i} P(h'_i \mid \mathbf{v}) h'_i \sum_{\mathbf{h}'_{-i}} P(\mathbf{h}'_{-i} \mid \mathbf{v}') \\
&= \sum_{h_i} P(h_i \mid \mathbf{v}) h_i - \sum_{\mathbf{v}'} P(\mathbf{v}') \sum_{h'_i} P(h'_i \mid \mathbf{v}) h'_i \\
&= \big[P(H_i = 0 \mid \mathbf{v}) \times 0 + P(H_i = 1 \mid \mathbf{v}) \times 1 \big] - \sum_{\mathbf{v}'} P(\mathbf{v}') \big[P(H_i = 0 \mid \mathbf{v}') \times 0 + P(H_i = 1 \mid \mathbf{v}') \times 1 \big] \\
&= P(H_i = 1 \mid \mathbf{v}) - \sum_{\mathbf{v}'} P(\mathbf{v}') P(H_i = 1 \mid \mathbf{v}') \\
&= \sigma(c_i + \sum_{k=1}^{d} w_{i,k} v_k) - \sum_{\mathbf{v}'} P(\mathbf{v}') \sigma(c_i + \sum_{k=1}^{d} w_{i,k} v'_k).
\end{aligned}$$

From the general formula for $\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial \theta}$,

$$\begin{aligned}
\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial b_i} &= \sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v}) v_i - \sum_{\mathbf{v}'} P(\mathbf{v}') \sum_{\mathbf{h}'} P(\mathbf{h}' \mid \mathbf{v}') v'_i \\
&= v_i \sum_{\mathbf{h}} P(\mathbf{h} \mid \mathbf{v}) - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_i \sum_{\mathbf{h}'} P(\mathbf{h}' \mid \mathbf{v}') \\
&= v_i - \sum_{\mathbf{v}'} P(\mathbf{v}') v'_i.
\end{aligned}$$

The results above allow maximizing $\ell(\boldsymbol{\theta} : \mathcal{D})$ with respect to $\boldsymbol{\theta}$ by gradient ascent. There is no guarantee that gradient ascent will find a global maxima, and the usual heuristics can be employed (momentum, for example).

For any parameter θ and observation $\mathbf{v} \in \mathcal{D}$, the computation of $\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial \theta}$ requires a term of the form

$$\sum_{\mathbf{v}'} P(\mathbf{v}' : \boldsymbol{\theta}) f(\mathbf{v}') = \mathbb{E}_{P(\mathcal{V}:\boldsymbol{\theta})}\big[f(\mathcal{V})\big],$$

for some factor $f : \text{Val}(\mathcal{V}) \to [0, 1]$. Because there are 2^d possible assignments to $\mathcal{V}$, this *expectation* is often computationally intractable. The remainder of this Section focuses on techniques employed to approximate $\mathbb{E}_{P(\mathcal{V}:\boldsymbol{\theta})}\big[f(\mathcal{V})\big]$.

Consider a data set $\mathcal{D}' = \{\mathbf{v}_i \mid i \in \{1, \dots, n'\}, \mathbf{v}_i \in \mathbb{R}^d\}$ composed of n' independent, identically distributed sample elements from $P(\mathcal{V} : \boldsymbol{\theta})$. By the law of large numbers, when $n' \to \infty$,

$$\frac{1}{n'} \sum_{\mathbf{v} \in \mathcal{D}'} f(\mathbf{v}) \to \mathbb{E}_{P(\mathcal{V}:\boldsymbol{\theta})}\big[f(\mathcal{V})\big].$$

Therefore, it is possible to estimate $\mathbb{E}_{P(\mathcal{V}:\boldsymbol{\theta})}\big[f(\mathcal{V})\big]$ by averaging $f(\mathcal{V})$ over a *large* sample $\mathcal{D}'$.

Gibbs sampling can be easily adapted for sampling from a restricted Boltzmann machine (Alg. 1), due to the specific independencies encoded by the graphical model. After enough time steps T (mixing time), this process is guaranteed to generate a sample element from the probability distribution $P(\mathcal{V}, \mathcal{H} : \boldsymbol{\theta})$ defined by the restricted Boltzmann machine. In general, T can be prohibitively large. The initial assignment $\mathbf{v}^{(0)}$ can be arbitrary, but *poor* choices lead to larger mixing times.

Algorithm 1 Gibbs sampling for restricted Boltzmann machines

Input: Conditional probability distributions $P(H_i \mid \mathcal{V} : \boldsymbol{\theta})$ and $P(V_j \mid \mathcal{H} : \boldsymbol{\theta})$, initial assignment $\mathbf{v}^{(0)}$, number of time steps T.

Output: Sequence $\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(T)}$ of assignments to $\mathcal{V}$.

```
1: for each $t$ in $1, \dots, T$ do
2:     for each $i$ in $1, \dots, D$ do
3:         $h_i^{(t)} \leftarrow$ sample from $P(H_i \mid \mathbf{v}^{(t-1)} : \boldsymbol{\theta})$
4:     end for
5:     for each $i$ in $1, \dots, d$ do
6:         $v_i^{(t)} \leftarrow$ sample from $P(V_i \mid \mathbf{h}^{(t)} : \boldsymbol{\theta})$
7:     end for
8: end for
```

Because $P(H_i = 1 \mid \mathbf{v}^{(t-1)}) = \sigma(c_i + \sum_{k=1}^{d} w_{i,k} v_k^{(t-1)})$, lines 2-4 in Alg. 1 are analogous to forward-passing $\mathbf{v}^{(t-1)}$ through a single layer of stochastic binary neurons.

There are two valid alternatives for collecting n' sample elements by Gibbs sampling: repeating the process n' times, and collecting each sample element after T steps; or collecting n' sample elements after T steps. In general, the second alternative requires a larger sample to achieve a similar-*quality* estimate.

After the sample $\mathcal{D}'$ is collected, it becomes possible to compute an estimate for $\frac{\partial \ell(\boldsymbol{\theta}:\mathcal{D})}{\partial \theta}$, for any θ. For more details on Gibbs sampling, see the notes on probabilistic graphical models.

Additional approximations based on Gibbs sampling are employed in practice. In k-step contrastive divergence, Gibbs sampling is started at $\mathbf{v}^{(0)} = \mathbf{v}$ to estimate $\frac{\partial \ell(\boldsymbol{\theta}:\mathbf{v})}{\partial \theta}$ for each $\mathbf{v} \in \mathcal{D}$. Gibbs sampling is performed for only k time steps ($T = k$), and $f(\mathbf{v}^{(T)})$ is used as an estimate of $\mathbb{E}_{P(\mathcal{V}:\boldsymbol{\theta})}\big[f(\mathcal{V})\big]$ (i.e., $n' = 1$).

In k-step persistent contrastive divergence, a number of *persistent* assignments $\mathbf{v}_1^{(0)}, \dots \mathbf{v}_p^{(0)}$ to $\mathcal{V}$ start at randomly chosen elements of $\mathcal{D}$. During each gradient ascent step, $T = k$ steps of Gibbs sampling are performed independently starting at each $\mathbf{v}_i^{(0)}$, and $\mathcal{D}' = \{\mathbf{v}_1^{(T)}, \dots \mathbf{v}_p^{(T)}\}$ is used to estimate $\mathbb{E}_{P(\mathcal{V}:\boldsymbol{\theta})}\big[f(\mathcal{V})\big]$. The last assignment $\mathbf{v}_i^{(T)}$ is maintained for the next gradient ascent step (i.e., $\mathbf{v}_i^{(0)} \leftarrow \mathbf{v}_i^{(T)}$).

Because computing the likelihood $\ell(\boldsymbol{\theta} : \mathcal{D})$ of the parameters $\boldsymbol{\theta}$ for the data set $\mathcal{D}$ is generally intractable, it is particularly hard to supervise restricted Boltzmann machine training. For practical advice, see [7]. For an introduction to extensions of this technique, see [8].

9 Unsupervised pre-training

An unsupervised pre-training method uses unlabeled data to initialize neural network parameters. After pre-training, networks are trained as usual (for classification or regression), using labeled data. Unsupervised pre-training is recommended when labeled data is relatively scarce, and can considerably improve the efficacy of deep neural networks [2, 8]. We present two methods for pre-training deep neural networks: stacking autoencoders and stacking restricted Boltzmann machines.

A (single hidden layer) autoencoder is a single hidden layer feedforward neural network trained to *predict* its own input. Consider the unlabeled data set $\mathcal{D} = \{\mathbf{x}_i \mid i \in \{1, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m\}$. Let $\mathbf{a}^{(l)}$ represent the layer l activation vector of a feedforward network that receives $\mathbf{x} = \mathbf{a}^{(1)}$ as input. An ideal autoencoder would predict $\mathbf{a}^{(3)} = \mathbf{x}$ on every input $\mathbf{x} \in \mathcal{D}$, and would also generalize well to unseen data. As usual, generalization can be evaluated by computing the cost function on a validation set.

The cost function and output layer of an autoencoder must be chosen appropriately. For a data set $\mathcal{D}$ composed of vectors in $\mathbb{R}^m$, the cost function can be the mean squared error and the output layer can be linear (identity). For vectors in $\{0, 1\}^m$, the natural choices are categorical cross-entropy and sigmoid output layer. Notice that a feedforward neural network with m linear hidden neurons and linear output layer could simply learn the identity function: $w_{i,j}^{(l)} = 1$ iff $i = j$, and 0 otherwise; $b_i^{(l)} = 0$.

If an autoencoder is trained successfully, each hidden layer activation $\mathbf{a}^{(2)}$ can be interpreted as an alternative representation of each $\mathbf{a}^{(1)} \in \mathcal{D}$, since it would preserve enough information to reconstruct $\mathbf{a}^{(1)}$ given appropriate parameters. In this case, it is said that $\mathbf{a}^{(1)}$ is encoded into $\mathbf{a}^{(2)}$, and that $\mathbf{a}^{(2)}$ is decoded into $\mathbf{a}^{(3)}$. By extension, the data set $\mathcal{D}$ is encoded into the data set $\mathcal{D}' = \{\mathbf{a}^{(2)} \mid \mathbf{a}^{(1)} \in \mathcal{D}\}$.

Autoencoders can be used for dimensionality reduction, by having a number of hidden neurons $N^{(2)}$ inferior to the dimension m of the input space. They can also be used to obtain *sparse* representations, where most elements of each code vector $\mathbf{a}^{(2)}$ are zero, by introducing a regularization term in the network cost function (sparse autoencoder). Finally, autoencoders can also be used to construct a representation that is robust to corruption of the original data, by training a network to predict each vector $\mathbf{x} \in \mathcal{D}$ from distinct corrupted versions of itself (denoising autoencoder).

In the context of unsupervised pre-training, a sequence of L of autoencoders can be trained using a greedy strategy. Given an unlabeled data set $\mathcal{D} = \mathcal{D}^{(0)}$, each autoencoder is trained using $\mathcal{D}^{(l-1)}$, and used to encode $\mathcal{D}^{(l-1)}$ into $\mathcal{D}^{(l)} = \{\mathbf{a}^{(2)} \mid \mathbf{a}^{(1)} \in \mathcal{D}^{(l-1)}\}$. After all L autoencoders are trained, they can be *stacked*. This is accomplished by using the hidden layer activation of each autoencoder as an input to the next autoencoder (thus eliminating their output layers). After stacking, a task-appropriate output layer can be introduced, and supervised training proceeds as usual, starting from the hidden layer parameters of the autoencoders. This procedure is motivated by the belief that this *hierarchy* of autoencoders is capable of extracting features that serve as a good starting point for supervised learning.

A similar greedy strategy can be employed to stack a sequence of L restricted Boltzmann machines. Given an unlabeled data set $\mathcal{D} = \mathcal{D}^{(0)}$ of independent, identically distributed sample elements from $P^{(0)}(\mathcal{V})$, each restricted Boltzmann machine is trained to represent a probability distribution $P^{(l)}(\mathcal{V}^{(l)}, \mathcal{H}^{(l)})$ over $\mathcal{D}^{(l-1)}$, and used to extract a sample $\mathcal{D}^{(l)}$ from $P^{(l)}(\mathcal{H}^{(l)})$. As usual, sampling can be attempted by Gibbs sampling or k-step contrastive divergence. After all L restricted Boltzmann machines are trained, they can be *stacked* by interpreting their weights $w_{j,k}^{(l)}$ and biases $c_j^{(l)}$ as parameters of a feedforward neural network with T hidden layers. As before, a task-appropriate output layer can be introduced, and supervised learning proceeds as usual.

License

References

[1] Nielsen, Michael. *Neural Networks and Deep Learning*, Available in http://neuralnetworksanddeeplearning.com, 2015.

[2] Hinton, Geoffrey. *Neural Networks for Machine Learning*, Available in http://www.coursera.org, 2012.

[3] Li, Fei-Fei and Karpathy, Andrej. *Convolutional Neural Networks for Visual Recognition*, Available in http://cs231n.github.io/convolutional-networks, 2015.

[4] Simonyan, Kare, and Zisserman, Andrew. *Very deep convolutional networks for large-scale image recognition*, Available in http://arxiv.org/abs/1409.1556, 2014.

[5] Graves, Alex. *Supervised Sequence Labelling with Recurrent Neural Networks*, 2012.

[6] Fischer, Asja and Igel, Christian. *An Introduction to Restricted Boltzmann machines.* CIARP, 2012.

[7] Hinton, Geoffrey. *A Practical Guide to Training Restricted Boltzmann Machines*, 2010.

[8] Bengio, Yoshua. *Learning deep architectures for AI.* Foundations and trends in Machine Learning, 2009.