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

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 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. 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:

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

Because $a_j^{(L)}$ is a differentiable function of $z_j^{(L)}$, and c is a differentiable function of $a_1^{(L)}, \ldots, a_{N^{(L)}}^{(L)}$ (and $z_j^{(L)}$ only affects c through $a_j^{(L)}$):

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

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:

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

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

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

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 alternatives 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 $i \in \{1, \dots, N\}$, $\mathbf{a} \in \mathbb{Z}^2$ 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 $i \in \{1, \dots, c\}$, $(j, k) \in \mathbb{Z}^2$, 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

7 Restricted Boltzmann machines

8 Deep auto encoders

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, 2015.

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

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

[5] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu and P. Kuksa. *Natural Language Processing (Almost) from Scratch*, Journal of Machine Learning Research, 2011.