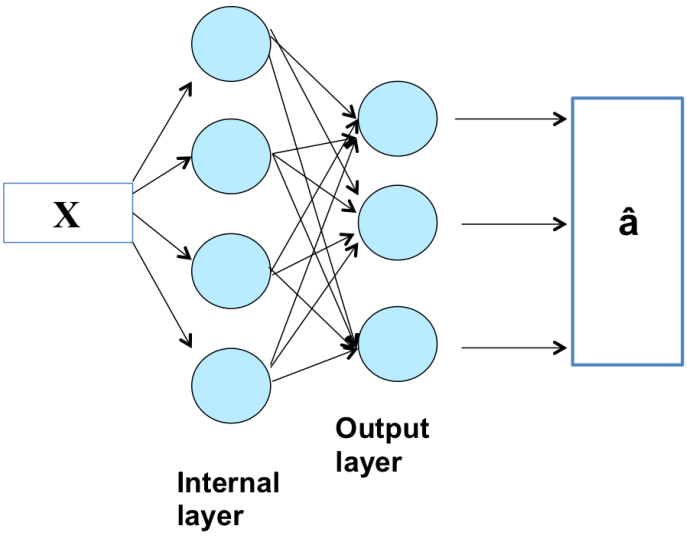
Notes on simple neural networks

Overview

The general aim is to build a model which given an input dataset, assigns a label to the dataset. The model is generally choosing from a small number of distinct labels. For example, provided image data, label the image as “dog”, “cat”, or “mouse”. A neural network is first “trained” – its fitting parameters are set so that an appropriately defined classification error (or cost function) is minimized. Below, I describe a simple neural network introducing some common jargon and definitions along the way.

A simple network



The input to the network is a column vector, $\mathbf{X} = [X_1, X_2, \ldots, X_N]^T$, and this vector is provided to each neuron in the inner (or ‘hidden’) layer. Each of these neurons computes a weighted sum of the vector elements and sends this sum and another fitting parameter (the bias) into a sigmoid function. The output from this function is the ‘activation’ (i.e. output) of the neuron. The activations from each neuron in the inner layer then form a column vector, $\mathbf{a} = [a_1, a_2, \ldots, a_M]^T$ where M is the number of neurons in the inner layer. This column vector is provided as input to each neuron in the output layer, and each of these neurons then computes an activation, and these activations, $\hat{\mathbf{a}} = [\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_P]^T$ are the final output from the network (here, *P* is the number of neurons in the output layer). There are now two points to consider: 1) how is the weighted sum constructed?, and 2) how are the weights/fitting parameters set?

The weighted (and biased) sum in the *ith* inner neuron is given by $z_i = \sum_{j=1}^{N} (w_{ij} X_j) + b_i$. The *M x N* matrix, w_{ij}, and the M-element column vector, b_i, are fitting parameters that are set during the training process. As stated above, each of these sums is fed into a sigmoid function to compute each inner neuron’s activation: $a_i = \frac{1}{1+exp(-z_i)}$.

These activations are then sent to each neuron in the output layer where weighted sums are again constructed, $\hat{z}_i = \sum_{j=1}^{M} (\hat{w}_{ij} a_j) + \hat{b}_i$ where a new *P x M* weight matrix, $\hat{w}_{ij}$, has been introduced along with the P-element column vector, $\hat{b}_i$. The output activations are then, $\hat{a}_i = \frac{1}{1+exp(-\hat{z}_i)}$. In the figure above, *M=4* and *P=3*.

Training

We now look at the training process which sets the $(N+1)*M + (M+1)*P$ fitting parameters. Let’s say we have *D* training datasets, $\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, \ldots, \mathbf{X}^{(D)}$ which have each been labeled with a P-element column vector, $\mathbf{Y}^{(k)}$, with $k = 1, 2, \ldots, D$. Then, we define a *cost function* which measures how well the network output, $\hat{\mathbf{a}}^{(k)}$ matches the labels, $\mathbf{Y}^{(k)}$. For example, using a least-squares error, we have the following optimization problem:

$$\text{Find } \left(w_{ij}, \hat{w}_{ij}, b_i, \hat{b}_i\right) \text{ such that C is minimized where, } C = \frac{1}{2D} \sum_{k=1}^{D} \sum_{i=1}^{P} \left[\left(\hat{a}_i^{(k)} - Y_i^{(k)} \right)^2 \right]$$

This problem can be readily solved with standard optimization methods when *D* is not too large, but for many practical classification problems, this is not a good assumption. For the handwritten digit classification problem, we were working with *28 x28* image matrices which corresponds to *N=784*. Taking *M=4* and *P=10*, we then have 3190 parameters which must be set. Given

the large problem size, two commonly-used optimization methods are: 1) *l-bfgs-b* which is a low-memory version of *bfgs* and is available in Scipy and 2) stochastic gradient descent which iterates in the direction of the negative gradient of C (while also only using a randomly subset of the *D* training datasets each iteration to reduce the computational cost). For both methods, the cost function, C, and its gradient, ∇C, need to be provided as input. Here, the gradient is a column vector containing the elements of $\frac{\partial C}{\partial \hat{b}_i}$, $\frac{\partial C}{\partial \hat{w}_{ij}}$, $\frac{\partial C}{\partial b_j}$, and $\frac{\partial C}{\partial w_{jl}}$. Examining the expression for C above, only the output activations depend on the fitting parameters, so the key is to obtain the derivatives of $\hat{\mathbf{a}}$ with respect to the two weight matrices and the two bias vectors $\left(w_{ij}, \hat{w}_{ij}, b_i, \hat{b}_i\right)$. First, let's consider the derivative for the outer layer bias:

$$\frac{\partial \hat{a}_i^{(k)}}{\partial \hat{b}_i} = \frac{\partial \hat{a}_i^{(k)}}{\partial \hat{z}_i^{(k)}} \frac{\partial \hat{z}_i^{(k)}}{\partial \hat{b}_i}$$

$$\frac{\partial \hat{z}_i^{(k)}}{\partial \hat{b}_i} = 1, \ \frac{\partial \hat{a}_i^{(k)}}{\partial \hat{z}_i^{(k)}} = \hat{a}_i^{(k)} \left(1 - \hat{a}_i^{(k)}\right).$$

The last equation is obtained by noting that,

$$f(z) = \frac{1}{1 + exp(-z)}, \ (1 - f) = \frac{exp(-z)}{1 + exp(-z)}$$

$$df/dz = \frac{exp(-z)}{(1 + exp(-z))^2} = f * (1 - f),$$

and finally we have,

$$\frac{\partial \hat{a}_i^{(k)}}{\partial \hat{b}_i} = \hat{a}_i^{(k)} \left(1 - \hat{a}_i^{(k)}\right). \tag{1}$$

Analogous results with the weight matrix, $\hat{w}_{ij}$, are constructed in a similar manner:

$$\frac{\partial \hat{a}_i^{(k)}}{\partial \hat{w}_{ij}} = \frac{\partial \hat{a}_i^{(k)}}{\partial \hat{z}_i^{(k)}} \frac{\partial \hat{z}_i^{(k)}}{\partial \hat{w}_{ij}}$$

$$\frac{\partial \hat{z}_i^{(k)}}{\partial \hat{w}_{ij}} = a_j^{(k)},$$

which leads to,

$$\frac{\partial \hat{a}_i^{(k)}}{\partial \hat{w}_{ij}} = a_j^{(k)} \hat{a}_i^{(k)} \left(1 - \hat{a}_i^{(k)}\right) \tag{2}$$

Moving now to the inner layer, we have:

$$\frac{\partial \hat{a}_i^{(k)}}{\partial b_j} = \frac{\partial \hat{a}_i^{(k)}}{\partial \hat{z}_i^{(k)}} \frac{\partial \hat{z}_i^{(k)}}{\partial a_j^{(k)}} \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial b_j}$$

$$\frac{\partial \hat{z}_i^{(k)}}{\partial a_j^{(k)}} = \hat{w}_{ij}$$

$$\frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} = a_j^{(k)} \left(1 - a_j^{(k)}\right), \ \frac{\partial z_j^{(k)}}{\partial b_j} = 1,$$

and combining the above equations,

$$\frac{\partial \hat{a}_i^{(k)}}{\partial b_j} = \hat{a}_i^{(k)} \left(1 - \hat{a}_i^{(k)}\right) \hat{w}_{ij} a_j^{(k)} \left(1 - a_j^{(k)}\right). \tag{3}$$

Finally, considering the 'inner' weight matrix,

$$\frac{\partial \hat{a}_i^{(k)}}{\partial w_{jl}} = \frac{\partial \hat{a}_i^{(k)}}{\partial \hat{z}_i^{(k)}} \frac{\partial \hat{z}_i^{(k)}}{\partial a_j^{(k)}} \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial w_{jl}}$$

$$\frac{\partial z_l^{(k)}}{\partial w_{jl}} = X_l^{(k)}$$

which give,

$$\frac{\partial \hat{a}_i^{(k)}}{\partial w_{jl}} = \hat{a}_i^{(k)} \left(1 - \hat{a}_i^{(k)}\right) \hat{w}_{ij} a_j^{(k)} \left(1 - a_j^{(k)}\right) X_l^{(k)}. \tag{4}$$

It is convenient to rewrite equations (1) - (4) in a more compact form. Defining $\hat{\gamma}^{(k)} = \hat{a}_i^{(k)} \left(1 - \hat{a}_i^{(k)}\right)$ and $\gamma_i^{(k)} = a_i^{(k)} \left(1 - a_i^{(k)}\right)$, the equations become,

$$\begin{aligned}
\frac{\partial \hat{a}_i^{(k)}}{\partial \hat{b}_i} &= \hat{\gamma}_i^{(k)} \\
\frac{\partial \hat{a}_i^{(k)}}{\partial \hat{w}_{ij}} &= a_j^{(k)} \hat{\gamma}_i^{(k)} \\
\frac{\partial \hat{a}_i^{(k)}}{\partial b_j} &= \hat{\gamma}_i^{(k)} \hat{w}_{ij} \gamma_j^{(k)} \\
\frac{\partial \hat{a}_i^{(k)}}{\partial w_{jl}} &= \hat{\gamma}_i^{(k)} \hat{w}_{ij} \gamma_j^{(k)} X_l^{(k)}
\end{aligned}$$

and for completeness, let's note that in the four equations above, $i \in \{1, \ldots, P\}$; $j \in \{1, \ldots, M\}$; $k \in \{1, \ldots, D\}$; and $l \in \{1, \ldots, N\}$. We now can provide expressions for the gradient of the cost function above, ∇C:

$$\begin{aligned}
\frac{\partial C}{\partial \hat{b}_i} &= \frac{1}{D} \sum_{k=1}^{D} \sum_{i=1}^{P} \left(\epsilon_i^{(k)} \frac{\partial \hat{a}_i^{(k)}}{\partial \hat{b}_i} \right) \\
\frac{\partial C}{\partial \hat{w}_{ij}} &= \frac{1}{D} \sum_{k=1}^{D} \sum_{i=1}^{P} \left(\epsilon_i^{(k)} \frac{\partial \hat{a}_i^{(k)}}{\partial \hat{w}_{ij}} \right) \\
\frac{\partial C}{\partial b_j} &= \frac{1}{D} \sum_{k=1}^{D} \sum_{i=1}^{P} \left(\epsilon_i^{(k)} \frac{\partial \hat{a}_i^{(k)}}{\partial b_j} \right) \\
\frac{\partial C}{\partial w_{jl}} &= \frac{1}{D} \sum_{k=1}^{D} \sum_{i=1}^{P} \left(\epsilon_i^{(k)} \frac{\partial \hat{a}_i^{(k)}}{\partial w_{jl}} \right) \\
\epsilon_i^{(k)} &= \left(\hat{a}_i^{(k)} - Y_i^{(k)} \right).
\end{aligned}$$

Computation

The optimization procedure consists of the following steps:

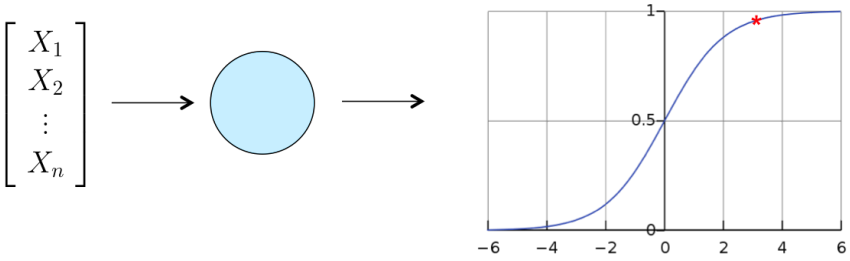
1. Guess fitting parameters, $\left(w_{ij}, \hat{w}_{ij}, b_i, \hat{b}_i\right)$
2. Compute all inner layer activations, $a_i^{(k)}$, $k \in \{1, \ldots, D\}$
3. Compute all output layer activations, $\hat{a}_i^{(k)}$, $k \in \{1, \ldots, D\}$
4. Compute cost function, C
5. Compute gradient components, $\frac{\partial C}{\partial \hat{b}_i}$, $\frac{\partial C}{\partial \hat{w}_{ij}}$, $\frac{\partial C}{\partial b_j}$, $\frac{\partial C}{\partial w_{ij}}$
6. Update fitting parameters and return to 2. if convergence criteria have not been satisfied

Testing (edited 6/11/18, definition of testing error corrected 11/11/18)

Usually, not all available labeled datasets are used in the training process. Instead a fraction (say, 1/3) is set aside to test the NN after it has been trained. The cost function can be evaluated using this test data to provide a 'testing error' which will generally be larger than the training error since the fitting parameters have been set to minimize the training error. See the NN code in Lab 3 for an example. The procedure for computing the testing error is as follows:

1. For all T images in the test dataset, compute the NN output, $\hat{\mathbf{a}}^{k)}$, $k \in \{1, \ldots, T\}$
2. Round each element of the output to the nearest integer (0 or 1)
3. Calculate $N_{correct}$: count the number of images for which the rounded output matches the T labels, $Y_{test}^{(k)}$
4. $1 - (N_{correct}/T)$ is then the testing error (**not** $(1 - N_{correct})/T$)

Single neuron model



In lecture, we first discussed a single neuron model (SNM) and then moved on to a network. Here, we take the reverse path; having introduced a (somewhat) general network, we will now recover the SNM. The basic idea behind the SNM is illustrated in the figure above with a given input vector being converted into a point where the the sigmoid function is evaluated. We proceed by discarding the output layer in our network model (P=0) and setting M=1. The M-element inner-layer activation vector, $\mathbf{a}$, then becomes a single activation, a, which is the SNM output:

$$z = \sum_{l=1}^{N} (w_l X_l) + b$$
$$a = \frac{1}{1 + exp(-z)},$$

and the weight matrix has become an N-element column vector, w_l, while b is now a scalar.

With a least-squares error, the optimization problem is,

$$\text{Find } (w_l, b) \text{ such that C is minimized where, } C = \frac{1}{2D} \sum_{k=1}^{D} \left(a^{(k)} - Y^{(k)} \right)^2$$

where the *label* corresponding to the *kth* input vector, $\mathbf{X}^{(k)}$, is the scalar, $Y^{(k)}$ and typically is zero or one for each k.

Expressions for constructing ∇C follow in a straightforward manner:

$$\frac{\partial C}{\partial b} = \frac{1}{D} \sum_{k=1}^{D} \left(\epsilon^{(k)} \frac{\partial a^{(k)}}{\partial b} \right)$$
$$\frac{\partial C}{\partial w_l} = \frac{1}{D} \sum_{k=1}^{D} \left(\epsilon^{(k)} \frac{\partial a^{(k)}}{\partial w_l} \right)$$
$$\epsilon^{(k)} = \left(a^{(k)} - Y^{(k)} \right).$$

The derivatives for the activations are found using the chain rule as before,

$$\frac{\partial a^{(k)}}{\partial b} = \gamma^{(k)}$$
$$\frac{\partial a^{(k)}}{\partial w_l} = X_l^{(k)} \gamma^{(k)}$$
$$\gamma^{(k)} = a^{(k)} \left(1 - a^{(k)} \right)$$

with $k \in \{1, \ldots, D\}; l \in \{1, \ldots, N\}$. It is left as an exercise to determine how the computation 'algorithm' for the network stated above is modified for the SNM.

Stochastic gradient descent (added 6/11/18)

In lecture, we considered gradient-based optimization methods starting from Newton's method which uses the cost function, its gradient, and its 2nd derivatives (collected in its Hessian) to iteratively move towards the minimizer of the cost. Gradient-descent methods neglect the curvature (in the Hessian) altogether and instead simply take steps in the direction of the negative of the gradient. The full gradient descent method for our NN model is:

$$\mathbf{f}^{[q+1]} = \mathbf{f}^{[q]} - \alpha \nabla C^{[q]}$$

where α is a parameter that sets how 'aggressively' the optimizer moves down the gradient, and the superscripts indicate the computation of the *q+1st* iteration using the fitting parameters from the *qth* iteration. All of the fitting parameters at a given iteration have been collected into the column vector, $\mathbf{f}^{(q)}$ In practice, for certain large classification problems, it is recognized that it is more efficient to compute the cost function (and gradient) using just one randomly chosen dataset at a time. The basic algorithm for one *epoch* of *stochastic gradient descent* (SGD) is:

1. guess fitting parameters (by, say, sampling them from a random normal distribution)
2. select a random training dataset, $k = \kappa$ (from 1,...,D)
3. compute ∇C with the sum over k truncated to $k = \kappa$ only
4. update the fitting parameters using the gradient descent formula above
5. Repeat steps 2-4 so a total of D steps are taken

Typically, SGD is run for several epochs, and this describes the algorithm in its most basic form. More sophisticated versions are typically used in machine learning (e.g. an adaptive learning rate, α is used).

Improving the cost function

To be added (not needed for homework 2)