where w_k are the singular values computed by SVD.

3.4 Sigma-clipping

As one last set of problems explored here, consider the need to fit the data *asymmetrically*; for example, we may want to fit a baseline, or an envelope of a signal.

Fortunately, there is no new formalism that needs to be developed; we will use linear least squares explained above in a special way.

1. Given data points (x_i, y_i), fit a model to all data.
2. Compute the residuals $\Delta_i = y_i - y(x_i|\boldsymbol{c})$ and the variance of the residuals, σ_0.
3. Given upper and lower clipping limits, η_{hi} and η_{lo}, remove all points that are above $\eta_{\mathrm{hi}}\sigma_0$ and below $\eta_{\mathrm{lo}}\sigma_0$.
4. Repeat the fit on the reduced number of data.
5. Continue until no further points are removed from the data.

This way, by controlling η_{lo} and η_{hi}, we control the part of the data-set that the model will fit.

4 Artificial neural networks

A range of problems in computational physics can be very difficult or even impossible to describe analytically, or the forward models take such a long time that any attempts at modeling are rendered impractical. In those cases we can consider an alternative: artificial neural networks.

An artificial neural network is a simple construct: it is a stack of interconnected *layers* (cf. Fig. 1). Each layer is an array of processing elements called *units*. These units propagate the signal between layers by *weighted connections*. The units perform non-linear *mapping* of input data to output parameters.

The input to the network is a measured or observed dataset – an array of sampled data points. These data are the stimulus of the *input* layer: each unit on the layer acquires a value of the given input array element. Once the input layer is populated, propagation to the *hidden* layer begins. The stimulus of the given unit on the hidden layer is a weighted sum of outputs from units on the current layer. This stimulus is then passed through a non-linear *activation function* that determines the extent of stimulation of the given unit. There is a connection from every unit on the input layer to every unit on the hidden layer, and each of these connections has a corresponding weight assigned to it. Once the signal has been propagated to the hidden layer, the propagation continues to the *output* layer: each unit acquires a value that is a weighted sum of outputs from units on the hidden layer, passed through the activation function.

Figure 1: ADD FIGURE! The topology of a three-layer Artificial Neural Network (ANN). Processing units (nodes) on the input layer are populated with input observations. A weighted sum of values on the input layer, $y_j = \sum_k w^H_{jk} i_k$, stimulates each unit, $j = 1 \dots l$, on the hidden layer. The amount of stimulus is determined by the non-linear activation function A^H_f, such as a sigmoid function, $A^H_f(y_j) = 1/[1 + \exp(-(y_j - \mu)/\tau)]$, with parameters μ and τ chosen in such a way that A^H_f is mapped onto the $[-1, 1]$ interval. Processing units on the hidden layer are then populated with $h_j = A^H_f(y_j)$. An analogous propagation takes place between the hidden layer and the output layer, using the same type of the activation function. The output layer of a trained network then contains the network's best guess at descriptive parameters of the data that have populated the input layer. The ANN is thus a *non-linear mapping* from observations to the physical parameters of the observed system.

The network thus *maps* its input to its output by propagating the stimulus via weighted connections that are passed through activation functions. Given the layer-to-layer connection weights, the propagation is basically a matter of summation and multiplication. The power comes from the ability of networks to provide *non-linear* mapping between their input and their output by using non-linear activation functions. The most commonly used activation functions are sigmoid curves, i.e. functions of the type $f(x) = 1/[1 + \exp(-(x - \mu)/\tau)]$.

The goal of ANNs is to have output layer values that have some representative significance of the input array. Since the output values depend on the connection weights, the task is to determine these weights. This is where *back-propagation* comes into play. Assume we have a sample of several thousand *exemplars* (input arrays for which we know the corresponding parameters) – obtained, for example, by computing theoretical models, or by using real data with reliable model solutions. For each exemplar we perform forward-propagation and compare the results output by the network with the true ones. We then *modify* the weights so that the discrepancy between the results is reduced for the whole sample. This iterative procedure is called the *training* or *machine learning* phase. It is the only computation-intensive block and it is performed only once. Once the weights are determined and the network reliably reproduces the expected results, the network is ready to *recognize* input never seen before. Hundreds of thousands of inputs can subsequently be processed in a matter of seconds on a single CPU.

The network described here, a basic three-layer back-propagating network (BPN), is remarkably robust for a diversity of non-linear problems. Different network topologies, such as multiple hidden layers, as well as more complicated connection strategies, advanced training approaches and other variations in general do not bring significant improvements to the basic model. For a thorough discussion on neural networks please refer to authoritative books such as Freeman & Skapura (1991).

4.1 Back-propagation

Back-propagation, as stated above, is the modification of weights to minimize the residuals between exemplar outputs and network-produced outputs on the output layer. As the mapping from input to output layer is multi-dimensional and non-linear, we resort to steepest descent.

The net input to the j-th unit on the hidden layer is:

$$\xi^h_{pj} = \sum_k w^h_{jk} i_k.$$

That net input is "activated" by the activation function $A_h(\xi^h_{pj})$. Similarly for the output layer:

$$\xi^o_{pi} = \sum_j w_{ij} h_j,$$

activated by $A_o(\xi^o_{pi})$. We want to minimize:

$$E_p = \frac{1}{2}\sum_p \delta^2_{pi}, \quad \text{where } \delta_{pi} = y_{pi} - o_{pi}.$$

To do that, we compute the (negative) gradient of E_p:

$$-\frac{\partial E_p}{\partial w_{ij}} = -\delta_{pi}\frac{\partial A_o(\xi^o_{pi})}{\partial \xi^o_{pi}}\frac{\partial \xi^o_{pi}}{\partial w^o_{ij}}. \tag{45}$$

Given the functional form of $A_o(\xi^o_{pi})$, the first derivative is simple to calculate; two common examples are linear and sigmoid:

$$A_o(\xi^o_{pi}) = \xi^o_{pi}, \quad \frac{\partial A_o(\xi^o_{pi})}{\partial \xi^o_{pi}} = 1;$$

$$A_o(\xi^o_{pi}) = \left(1 + e^{-\xi^o_{pi}}\right)^{-1}, \quad \frac{\partial A_o(\xi^o_{pi})}{\partial \xi^o_{pi}} = \xi^o_{pi}(1 - \xi^o_{pi}) \equiv o_{pi}(1 - o_{pi}).$$

The second derivative is also simple to calculate:

$$\frac{\partial \xi^o_{pi}}{\partial w^o_{ij}} = \frac{\partial}{\partial w^o_{ij}}\sum_j w_{ij}h_j = h_j.$$

Thus,

$$-\frac{\partial E_p}{\partial w_{ij}} = -\delta_{pi}A'_o(\xi^o_{pi})h_j \equiv \delta^o_{pi}h_j.$$

Here we introduced $\delta^o_{pi} \equiv \delta_{pi}A'_o(\xi^o_{pi})$. Then we correct the weights by taking a step along the gradient:

$$w^o_{ij} \mapsto w^o_{ij} + \eta\delta^o_{pi}h_j. \tag{46}$$

We do the exact same procedure for adjusting weights from the input layer to the hidden layer.

To train the neural network, we thus:

1. apply the input vector $\boldsymbol{i} = (i_1, \ldots, i_N)$ to the N input units;
2. calculate the net input values to the hidden layer units as:
$$h_j = A_h\left(\sum_k w^h_{jk} i_k\right), \quad \text{where } A_h(x) = \left(1 - e^{-(x-\mu)/\tau}\right)^{-1}; \tag{47}$$
3. calculate the net input values to the output layer as:
$$o_i = A_o\left(\sum_j w^o_{ij} h_j\right), \quad \text{where } A_o(x) = \left(1 - e^{-(x-\mu)/\tau}\right)^{-1}; \tag{48}$$
4. calculate the error terms for the output units:
$$\delta^o_i = (y_i - o_i) A'_o\left(\sum_j w^o_{ij} h_j\right), \tag{49}$$
where y_i are known values of parameters;
5. calculate the error terms for the hidden units:
$$\delta^h_j = A'_h\left(\sum_k w^h_{jk} i_k\right) \sum_i \delta^o_i w^o_{ij}; \tag{50}$$
6. update weights on the output layer:
$$w^o_{ij} \mapsto w^o_{ij} + \eta \delta^o_i A_o\left(\sum_j w^o_{ij} h_j\right), \tag{51}$$
where η is the learning rate parameter;
7. update weights on the hidden layer:
$$w^h_{jk} \mapsto w^h_{jk} + \eta \delta^h_j i_j; \tag{52}$$
8. calculate the overall error term:
$$E_p = \frac{1}{2} \sum_i \delta^2_i \quad \text{for the } p\text{-th exemplar;} \tag{53}$$
9. repeat all steps for all P exemplars.

The idea is to keep training the network for as long as E_p are decreasing and stop once they dip below a certain threshold.

So far we only talked about neural networks used for regression; another frequent application is classification.