Supervised Learning in Neural Networks

April 11, 2010

1 Introduction

Supervised Learning scenarios arise reasonably seldom in real life. They are situations in which an agent (e.g. human) performs many similar tasks, or many similar steps of a longer task, and receives *frequent and detailed feedback*: after each step, the agent learns a) whether or not her action was correct, and, in addition, b) the action that should have been performed. In real life, this type of feedback is not only rare, but annoying. Few people wish to have their every move corrected.

In Machine Learning (ML), however, this learning paradigm has generated a lot of research interest for decades. The classic tasks involve classification, wherein the system is given a large data set, with each case consisting of many features plus a classification. For example, the features might be meteorological factors such as temperature, humidity, wind velocity, etc., while the class could be the predicted precipitation level (high, medium, low, or zero) for the next day.

The supervised-learning classifier system must then learn to predict future precipitation when given a vector of meteorological features. The system receives the input features for a case and produces a precipitation prediction, whose accuracy can be assessed by a comparison to the known precipitation attached to that case. Any discrepancy between the two can then serve as an error term for modifying the system (so as to make better predictions in the future). Classification tasks are a staple of Machine Learning, and artificial neural networks are one of several standard tools (including decision-tree learning, case-based reasoning and Bayesian methods) used to tackle them.

This chapter investigates the main ANN approach to supervised learning: feed-forward networks with back-propgation learning. This technique covers a large percentage of all practical applications of ANNs, whose surge in popularity as automated learning tools stemmed directly from the invention of the backpropgation algorithm in the 1980's.

2 The Backpropagation Algorithm: Gradient Descent Training of ANNs

The basic idea behind backpropagation learning is to gradually adjust the weights of an artificial neural network (ANN) so as to reduce the error between the actual and desired outputs on a series of training cases. Each case is typically a pair, (d_i, r_i), indicating an element of the mapping between a domain and a

range for some (yet unknown) function. By training the ANN to reduce this error, one effectively *discovers* the function.

Figure 1 summarizes the basic process, wherein training cases are presented to the ANN, one at a time. The domain value, d_i, of each case is encoded into activation values for the neurons of the input layer (right). These values are then propagated through the network to the output layer, whose values are then decoded into a value of the range, r*, which is compared to the desired range value, r_i. The difference between these two values constitutes the error term.

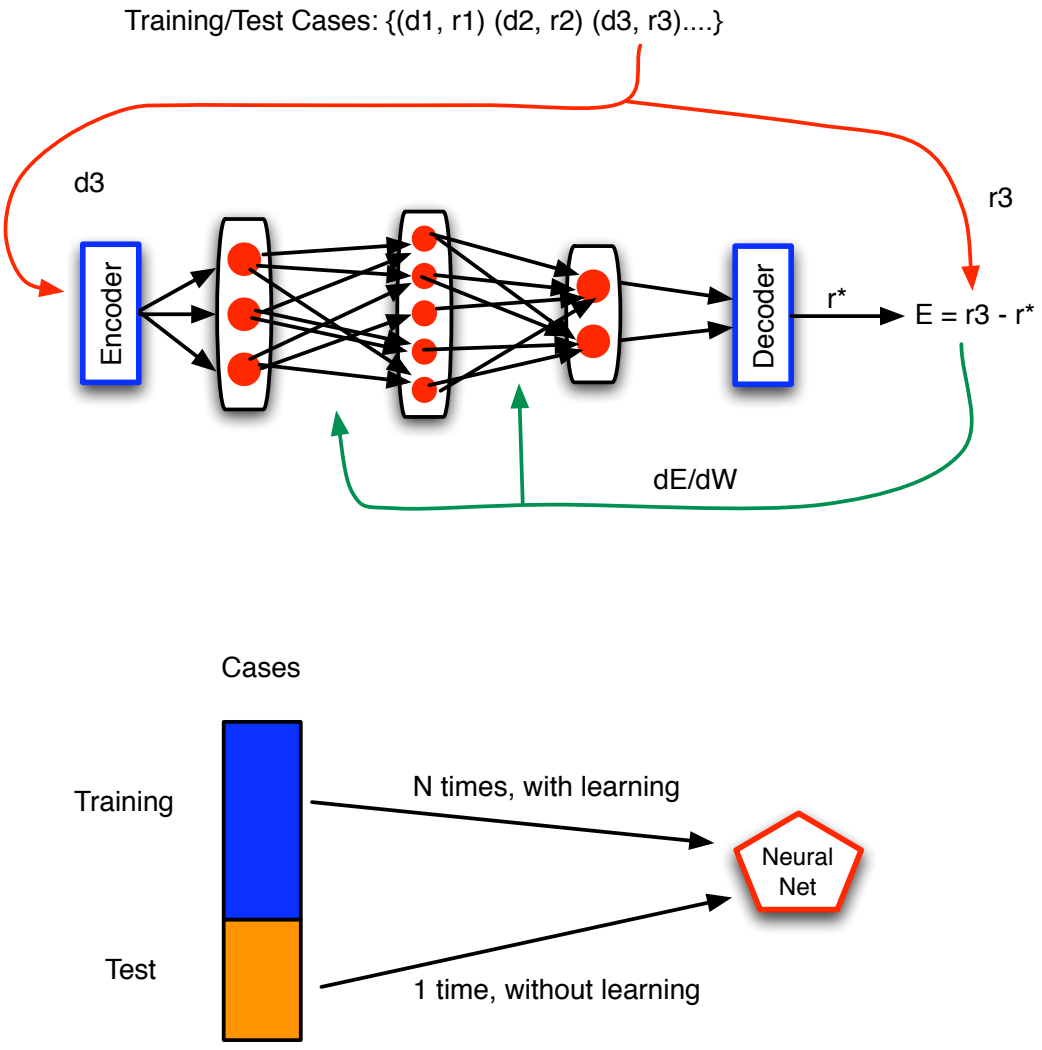

Figure 1: The essence of the backpropagation algorithm: During training, cases are encoded and sent through the network. The resulting outputs are compared to desired outputs to compute an error term, E, which is then used to compute changes to the weights of the network so as to reduce E.

3 Gradient Descent Learning For Perceptrons

The whole idea behind gradient descent is to gradually, but consistently, decrease the output error by adjusting the weights. The trick is to figure out HOW to adjust the weights. Intuitively, we know that if a change in a weight will increase (decrease) the error, then we want to decrease (increase) that weight. Mathematically, this means that we look at the derivative of the error with respect to the weight: $\frac{\partial E}{\partial w_{ij}}$, which represents the change in the error given a unit change in the weight.

Once we find this derivative, we will update the weight via the following:

$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} \tag{1}$$

This essentially represents the distance times the direction of change. The distance, η, is a standard parameter in neural networks and often called the learning rate. In more advanced algorithms, this rate may gradually decrease during the epochs of the training phase.

If we update all the weights using this same formula, then this amounts to moving in the direction of steepest descent along the error surface - hence the name, gradient descent - as shown in Figure 2.

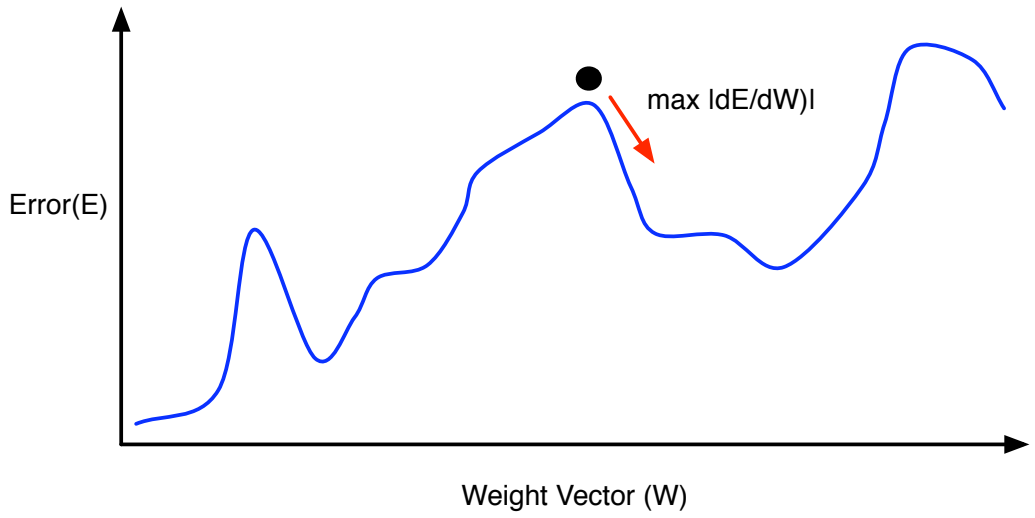

Figure 2: Abstract depiction of gradient descent as search in an error landscape, where the choice is among alternative changes to the ANN's complete weight vector. That change which gives the maximum decrease (red arrow) from the current location (black dot) is preferred. Note that in this case, the maximum decrease, a *greedy* choice to move to the right, does not lead in the direction of the global error minimum, which resides on the left side of the graph.

The above equation is easy, and it captures our basic intuitions: decrease (increase) a weight that positively (negatively) contributes to the error. Unfortunately, computing $\frac{\partial E}{\partial w_{ij}}$ is not so trivial.

For the following derivation, the perceptron depicted in Figure 3 will be used.

First we need to compute the error, and in the case of a perceptron or a row of independent perceptrons,

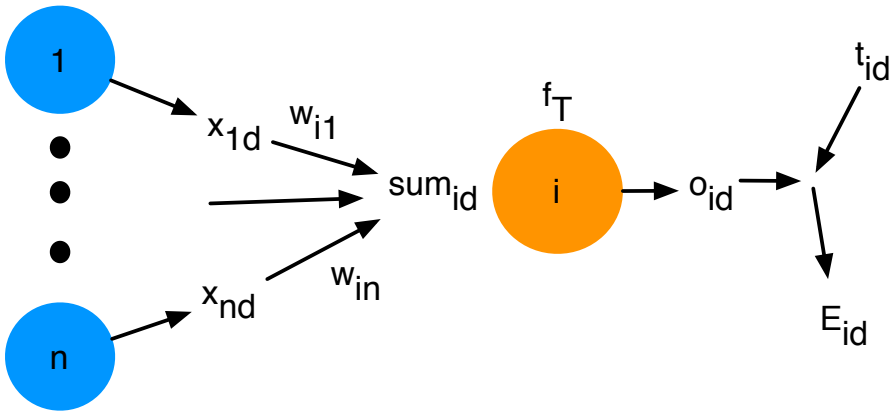

Figure 3: Simple perceptron with n weighted input lines

we can focus on the error at any particular output node, i. A standard metric is the sum of squared errors (SSE):

$$E_i = \frac{1}{2}\sum_{d\in D}(t_{id} - o_{id})^2 \tag{2}$$

where t_{id} and o_{id} are the desired/target and actual outputs, respectively, at node i on example data instance d.

So, to find $\frac{\partial E_i}{\partial w_{ij}}$, we compute:

$$\frac{\partial\left(\frac{1}{2}\sum_{d\in D}(t_{id} - o_{id})^2\right)}{\partial w_{ij}} \tag{3}$$

Taking the derivative inside the summation and using standard calculus:

$$\frac{1}{2}\sum_{d\in D} 2(t_{id} - o_{id})\frac{\partial(t_{id} - o_{id})}{\partial w_{ij}} = \sum_{d\in D}(t_{id} - o_{id})\frac{\partial(-o_{id})}{\partial w_{ij}} \tag{4}$$

The t_{id} term disappears from the derivative, since $\frac{\partial t_{id}}{\partial w_{ij}} = 0$, i.e. the target value is completely independent of the weight. This makes sense, since the target is set from outside the system.

Now, the output value o_{id} is equal to the transfer function for the perceptron, f_T, applied to the sum of weighted inputs to the perceptron (on example instance d), sum_{id}. So we can rewrite as:

$$\sum_{d\in D}(t_{id} - o_{id})\frac{\partial(-f_T(sum_{id}))}{\partial w_{ij}} \tag{5}$$

where:

$$sum_{id} = \sum_{k=1}^{n} w_{ik} x_{kd} \tag{6}$$

Here, summing over the k means summing over the n inputs to node i. That is, we sum the weighted outputs of all of i's suppliers.

Using the Chain Rule from calculus, which states that:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g(x)} \times \frac{\partial g(x)}{\partial x} \tag{7}$$

we can calculate the derivative of the transfer function with respect to the weight as follows:

$$\frac{\partial (f_T(sum_{id}))}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} \tag{8}$$

Now, the first term on the right-hand-side will vary, depending upon the transfer function. Shortly, we will compute it for a few different functions. But first, we can (easily) compute the second term of the product:

$$\frac{\partial sum_{id}}{\partial w_{ij}} = \frac{\partial \left(\sum_{k=1}^{n} w_{ik} x_{kd}\right)}{\partial w_{ij}} = \frac{\partial \left(w_{i1}x_{1d} + w_{i2}x_{2d} + \ldots + w_{ij}x_{jd} + \ldots + w_{in}x_{nd}\right)}{\partial w_{ij}} \tag{9}$$

$$= \frac{\partial (w_{i1}x_{1d})}{\partial w_{ij}} + \frac{\partial (w_{i2}x_{2d})}{\partial w_{ij}} + \ldots + \frac{\partial (w_{ij}x_{jd})}{\partial w_{ij}} + \ldots + \frac{\partial (w_{in}x_{nd})}{\partial w_{ij}} = 0 + 0 + \ldots + x_{jd} + \ldots + 0 = x_{jd} \tag{10}$$

This makes perfect sense: the change in the sum given a unit change in a particular weight, w_{ij}, is simply x_{jd}.

Next, we calculate $\frac{\partial f_T(sum_{id})}{\partial sum_{id}}$ for different transfer functions, f_T.

3.1 The Identity Transfer Function

First, if f_T is the identity function, then $f_T(sum_{id}) = sum_{id}$, and thus:

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = 1 \tag{11}$$

So in that simple case:

$$\frac{\partial(f_T(sum_{id}))}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} = 1 \times x_{jd} = x_{jd} \tag{12}$$

So putting everything together, for the identity transfer function, the derivative of the error with respect to weight w_{ij} is:

$$\frac{\partial E_i}{\partial w_{ij}} = \frac{\partial \left(\frac{1}{2}\sum_{d\in D}(t_{id} - o_{id})^2\right)}{\partial w_{ij}} = \sum_{d\in D}(t_{id} - o_{id})\frac{\partial(-o_{id})}{\partial w_{ij}} = \sum_{d\in D}(t_{id} - o_{id})\frac{\partial(-f_T(sum_{id}))}{\partial w_{ij}} \tag{13}$$

$$= -\sum_{d\in D}\left((t_{id} - o_{id})\frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}}\right) = -\sum_{d\in D}((t_{id} - o_{id})(1)x_{jd}) = -\sum_{d\in D}(t_{id} - o_{id})x_{jd} \tag{14}$$

Thus, the weight update for a neural network using identity transfer functions is:

$$\Delta w_{ij} = -\eta\frac{\partial E_i}{\partial w_{ij}} = \eta\sum_{d\in D}(t_{id} - o_{id})x_{jd} \tag{15}$$

This weight-update rule (that either ignores the transfer function or assumes it to be the identity function) is known as the Widrow-Huff or *delta* rule. One of the early types of simple neural units, called an Adaline, used the delta rule with neurons that had step transfer functions. So even though the step function is not everywhere differentiable, by treating it like an identity function, you can still get a useful estimate of $\frac{\partial E_i}{\partial w_{ij}}$.

3.2 The Sigmoidal Transfer Function

The sigmoidal function is very popular for neural networks, because it performs very similar to a step function, but it is everywhere differentiable. Thus, we can compute $\frac{\partial f_T(sum_{id})}{\partial sum_{id}}$ for the sigmoidal, although we cannot for a standard step function (due to the discontinuity at the step point).

The standard form of the sigmoidal is:

$$f_T(sum_{id}) = \frac{1}{1 + e^{-sum_{id}}} \tag{16}$$

So

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = \frac{\partial\left((1 + e^{-sum_{id}})^{-1}\right)}{\partial sum_{id}} = (-1)\frac{\partial(1 + e^{-sum_{id}})}{\partial sum_{id}}(1 + e^{-sum_{id}})^{-2} \tag{17}$$

$$= (-1)(-1)e^{-sum_{id}}(1 + e^{-sum_{id}})^{-2} = \frac{e^{-sum_{id}}}{(1 + e^{-sum_{id}})^2} \tag{18}$$

But, this is the same as:

$$f_T(sum_{id})(1 - f_T(sum_{id})) = o_{id}(1 - o_{id}) \tag{19}$$

Summarizing the result so far:

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = \frac{e^{-sum_{id}}}{(1 + e^{-sum_{id}})^2} = f_T(sum_{id})(1 - f_T(sum_{id})) = o_{id}(1 - o_{id}) \tag{20}$$

Now we can compute $\frac{\partial E_i}{\partial w_{ij}}$ for neural networks using sigmoidal transfer functions:

$$\frac{\partial E_i}{\partial w_{ij}} = \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial(-f_T(sum_{id}))}{\partial w_{ij}} = -\sum_{d \in D} \left((t_{id} - o_{id}) \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} \right) = -\sum_{d \in D} (t_{id} - o_{id}) o_{id}(1 - o_{id}) x_{jd} \tag{21}$$

So, the weight update for networks using sigmoidal transfer functions is:

$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} = \eta \sum_{d \in D} (t_{id} - o_{id}) o_{id}(1 - o_{id}) x_{jd} \tag{22}$$

3.3 Batch versus Incremental Updating

Notice that the previous update rules for w_{ij}, for both the identity and sigmoidal functions, involve batch processing in that we compute the total error for all the training instances ($d \in D$) before we update the weights.

An alternative is to update the weights after EACH example has passed through the network. In this case, the update rules are:

$$\eta(t_{id} - o_{id}) x_{jd} \tag{23}$$

for the identity function, and for the sigmoidal:

$$\eta(t_{id} - o_{id}) o_{id}(1 - o_{id}) x_{jd} \tag{24}$$

In these equations, the *error terms* associated with each **node**, i, on instance d are:

$$(t_{id} - o_{id}) \tag{25}$$

for the identity function, and for the sigmoidal:

$$(t_{id} - o_{id})o_{id}(1 - o_{id}) \tag{26}$$

Clearly, one should choose a much lower value of the learning rate, η, for incremental than for batch processing.

For batch processing with a single layer of output perceptrons, and thus only one layer of weights to learn, it suffices to simply sum the products of x_{jd} and the error term over all the training data, and then to add Δw_{ij} to w_{ij} at the end of the training epoch.

The main drawback of incremental updating is that the final weight values can be dependent upon the **order** of presentation of the examples. As explained in a later section, the computational efforts involved in incremental and batch processing are approximately equal.

4 Gradient Descent for Multi-Layer Neural Networks

For multi-layer networks, the relationship between the error term and ANY weight ANYWHERE in the network needs to be calculated. This involves propagating the error term at the output nodes backwards through the network, one layer at a time. At each layer, m, an error term (similar to those discussed above) is computed for each node,i, in the layer. Call this δ_{id}. It represents the negative of the effect of sum_{id} on the output error for example d, that is:

$$\delta_{id} = -\frac{\partial E_d}{\partial sum_{id}} \tag{27}$$

Then, going backwards to layer m-1, for each node, j, in that layer, we want to compute $\frac{\partial E_d}{\partial sum_{jd}}$. This is accomplished by computing the product of:

- the influence of j's input sum upon j's output: $\frac{\partial f_T(sum_{jd})}{\partial sum_{jd}}$, which is just $o_{jd}(1 - o_{jd})$ for a sigmoidal f_T.
- the sum of the contributions of j's output value to the total error via each of j's downstream neighbors.

In this section, the variable o is used to denote the output values of all nodes, not just those in the output layer.

The relationships between the different partial derivatives is easiest to see with the diagram in Figure 4. Here, the goal is to find the influence of sum_{jd} upon E_d by summing its influence along each path. For example, the effect along the upper route in Figure 4 is:

$$\frac{\partial o_{jd}}{\partial sum_{jd}} \times \frac{\partial sum_{1d}}{\partial o_{jd}} \times \frac{\partial E_d}{\partial sum_{1d}} \tag{28}$$

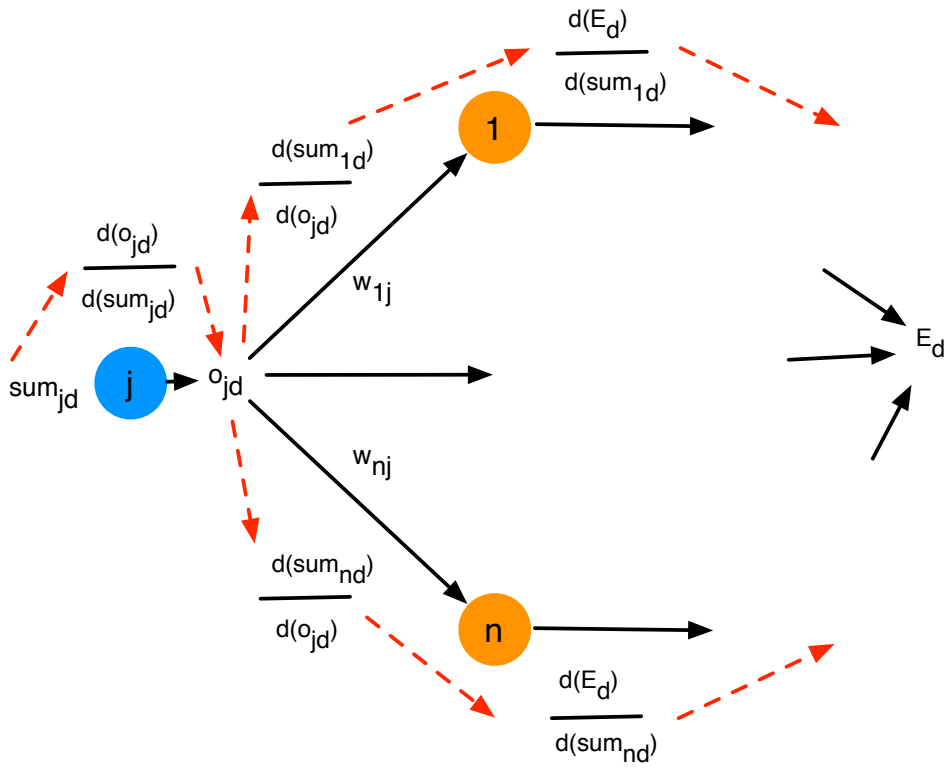

Figure 4: Multi-layer network displaying basic backpropagation terms and their relationships

Thus, the complete equation for computing node j's contribution to the error is:

$$\frac{\partial E_d}{\partial sum_{jd}} = \frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} \frac{\partial sum_{kd}}{\partial o_{jd}} \frac{\partial E_d}{\partial sum_{kd}} \tag{29}$$

Just as $\frac{\partial sum_{kd}}{\partial w_{kj}} = o_{jd}$, since most of the terms in the summation are zero (as shown in equations 9 and 10), we also have:

$$\frac{\partial sum_{kd}}{\partial o_{jd}} = w_{kj} \tag{30}$$

Substituting equations 27 and 30 into equation 29 yields :

$$\delta_{jd} = -\frac{\partial E_d}{\partial sum_{jd}} = -\frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} w_{kj}(-\delta_{kd}) = \frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} w_{kj}\delta_{kd} \tag{31}$$

This illustrates how the error terms at level m-1 (i.e. the δ_{jd}) are based on backpropagated error terms from level m (i.e., the δ_{kd}), with each layer-m error adjusted by the weight along the arc from node j to node k.

Assuming sigmoidal transfer functions:

$$\frac{\partial o_{jd}}{\partial sum_{jd}} = \frac{\partial f_T(sum_{jd})}{\partial sum_{jd}} = o_{jd}(1 - o_{jd}) \tag{32}$$

Thus,

$$\delta_{jd} = o_{jd}(1 - o_{jd}) \sum_{k=1}^{n} w_{kj}\delta_{kd} \tag{33}$$

This provides an operational definition for implementing the recursive backpropagation algorithm for networks using sigmoidal transfer functions:

1. For each output node, i, compute its error term using:

$$\delta_{id} = (t_{id} - o_{id})o_{id}(1 - o_{id}) \tag{34}$$

2. Working backwards, layer by layer, compute the error terms for each internal node, j, as:

$$\delta_{jd} = o_{jd}(1 - o_{jd}) \sum_{k=1}^{n} w_{kj}\delta_{kd} \tag{35}$$

   where the k are the downstream neighbor nodes.

Once we know $\frac{\partial E_d}{\partial sum_{id}} = -\delta_i$, the computation of $\frac{\partial E_d}{\partial w_{ij}}$ is quite easy. This stems from the fact that the only effect of w_{ij} upon the error is via its effect upon sum_{id}:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial sum_{id}}{\partial w_{ij}} \times \frac{\partial E_d}{\partial sum_{id}} = \frac{\partial sum_{id}}{\partial w_{ij}} \times (-\delta_{id}) = -o_{jd}\delta_{id} \tag{36}$$

So, given an error term, δ_i, for node i, the update of w_{ij} for all nodes j feeding into i is simply:

$$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}} = -\eta(-o_{jd}\delta_{id}) = \eta\delta_{id}o_{jd} \tag{37}$$

Obviously, the procedure is similar for other types of transfer functions, such as the linear or inverse hyperbolic tangent function. It is only $\frac{\partial f_T(sum_{jd})}{\partial sum_{jd}}$ that will vary.

5 Batch Processing for Multilayer Neural Networks

As shown in equations 34 and 35, when performing incremental weight updates, the error terms for nodes are a function of the output values of those nodes on a particular training case. Then, as shown in equations 36 and 37, the contribution of w_{ij} to the error (which determines Δw_{ij}) is also a function of the output value of node j on that same training case.

Hence, for batch processing, it will not suffice to update the node error terms after each training case and then, at the end of the epoch, update the weights based on the node errors. Instead, we need to keep track of the Δw_{ij} after each training case, *but we do not actually update the weight until the end of the epoch.* So we sum the recommended weight changes after each training instance, but we only modify the weight at epoch's end. Formally, we are computing:

$$\sum_{d \in D} \Delta_d w_{ij} \tag{38}$$

Notice that the recommended changes can be positive or negative, so they may partially cancel one another out.

Unfortunately, this means that batch processing does not save computational effort, since we are essentially performing full backpropagation after each training instance. Although we do not update w_{ij} on each instance, we update the sum of the changes (equation 38), which is just as much work, computationally.

However, practically speaking, it makes more sense to change w_{ij} once, at the end of an epoch, than to raise and lower it many times during the course of that same epoch. With incremental updating, the ANN can be sensitive to the order in which examples are presented, which is normally considered a weakness. Since the batch approach modifies weights only at the end of an epoch, based on the average information from each of the cases, this order sensitivity is eliminated.

However, there are advantages to the incremental approach, as discussed by Haykin [6]. To avoid order sensitivity while maintaining an incremental update scheme, it suffices to randomly permute the presentation

order before each epoch. This has the added advantage of lending stochasticity to the learning procedure, and this extra *jiggle* can be just enough to avoid stagnation in a local minimum.

6 Practical Tips

In practice, backpropagation is as much an art as a science. The user typically needs to try many combinations of parameter settings before finding that which best suits the current task. Few concrete rules exist to cover a wide range of application domains. There are many factors to consider, and we touch on a few of them below. A host of additional tips can be found in [3, 6, 10].

6.1 Hidden layers and nodes

Although there are some theoretical results, there is no hard and fast rule as to how many hidden layers and nodes in each layer are required for a given problem. There are a few considerations.

1. If all of your training and test cases are, in fact, linearly separable, then you can get by without a hidden layer. Unfortunately, most real-life data sets are not linearly separable.
2. If the data are not linearly separable, then, in theory, a **single** hidden layer is sufficient for learning the data. However, the exact number of nodes in that layer cannot be accurately calculated ahead of time. Also, the practially-best solution may involve multiple hidden layers.
3. The more layers and nodes in an ANN, the more connections, and thus the more weights that need to be learned. Adding more connections than necessary can easily increase training time well above that needed to solve the problem.
4. Excess connections increase the risk of overfitting the ANN weight vector to the training set. Thus, the ANN may perform perfectly on the training set but poorly on the test cases. This indicates that the ANN has *memorized* the training cases but cannot *generalize* from them to handle new cases. This is highly undesirable. To prevent this (in cases where a perceived demand for many connections exists) it often helps to halt training before the total error gets too close to 0.

6.2 Ranges for desired values

When converting desired outputs into target activations for the output nodes of an ANN, it is wise to use a restricted range of values. This pertains to ANNs that use any kind of sigmoidal activation function.

For example, if the sigmoid's output range is [0, 1], then it pays to use a subrange such as [0.1, 0.9] for the desired output values. The problem lies in the nature of the sigmoid function, which has **almost** the same minimum or maximum value for large sections of the domain, i.e., the area far from the threshold point. In trying to attain this asymptote of the sigmoid, the backpropagation algorithm will often increase n's incoming weights indefinitely, and these weight increases can propagate backwards via equation 33. Once weights become very large, backpropagation has a hard time reducing them quickly, or at all. The whole system can easily spiral out of control, with most weights approaching infinity and most sigmoids becoming *saturated*, i.e. having weighted sum inputs that push them to the extreme ends of their range.

By using target values well below the asymptotic limit of the sigmoid, one insures that targets can be achieved exactly and weights will remain more stable.

6.3 Ranges for activation levels

For any application, it is important to decide whether low values of a particular factor, such as an input parameter, should have either a) a low effect upon downstream nodes, or b) the **opposite** effect of a high value. In the former case, your nodes should output in the range [0,1], while the latter case probably calls for a [-1, 1] range.

6.4 Information content of nodes

One should exercise caution when using *tricky* encodings of information on the input and output ends of an ANN, even though these can yield more compact networks (i.e., those with fewer nodes and connections).

For example, if the color of an automobile is an input parameter to an ANN for assessing insurance risk, then a compact encoding might involve a single input neuron, n, whose activation value would indicate everything from dark colors (low activation levels) to bright colors (high activation levels).

Keep in mind that the goal of the ANN is to learn a proper, **fixed** set of weights emanating from n that will reduce total error and thus give good predictions of insurance risk based on color (and other input variables).

Whatever these weights may be, they represent the total effect of color upon all downstream nodes. And the activation level of n will further modulate that effect, but changes to that activation are severely constrained as to possible changes in effect. In essence, they are constrained to linear behavior. If the activation level goes up, then the absolute value of the effect will also increase.

So only if there does indeed exist a linear relationship between the color and the net effect (on insurance risk), can this encoding work effectively. However, it could easily be the case that both very bright colors (such as hot red) and dark colors (black) are typical colors of cars owned by drivers with a *street racer* mentality. In fact, there is a strong correlation in the United States between these hot colors and speeding tickets. I am only speculating about the black colored cars for the purpose of this example. Conversely, colors in the middle of the spectrum, such as pale blue, are more typical *family car* colors.

The problem is that if both black and red have effects that trigger a high insurance risk, then by tuning weights to realize that influence, one cannot avoid producing weights that will also give blue a similar effect. In this example, the weights would have to be very high to allow both a low color input (black) to achieve a similar influence to that of a high-color input (red).

In this case, it is wiser to use k input nodes, where each represents a single color. Then, backpropagation is free to tune the individual effects of each color (by modifying its outgoing weights) without being hampered by any artificial connections between the colors (as imposed by the more compact representation).

Conversely, if the input factor truly does have a linear effect upon the output factors, for example, if weight is an input condition and risk of heart-attack is an output class, then compressing the range of inputs into a single input node makes more sense.

The same can be said about output nodes and the information encoded in them. In this case, the input arcs

to an output node represent the net effect of the ANN's information upon the outputs. Once again, a linear relationship holds such that the sum of the weighted inputs has a non-decreasing effect upon the output. [1].

Once those weights are fixed by learning, they enforce a linear relationship between any given node, n_1 in the previous layer and a particular output node, n_2. Thus, the target concepts to which activation levels of n_2 correspond should have a linear relationship to the information represented by n_1. If not, then the ANN will not be able to determine an optimal mapping between inputs and outputs.

The above advice is heuristic in nature, not absolute. A tricky encoding here or there may not destroy ANN performance, but it is important to be aware of the gains (compact ANNs) and losses (inability to properly differentiate certain cases) associated with witty encoding schemes.

6.5 Momentum

Backpropagation can easily get stuck at a local minimum, such as that pictured in Figure 5. There, all gradients point upward, toward higher error, so all progress stagnates.

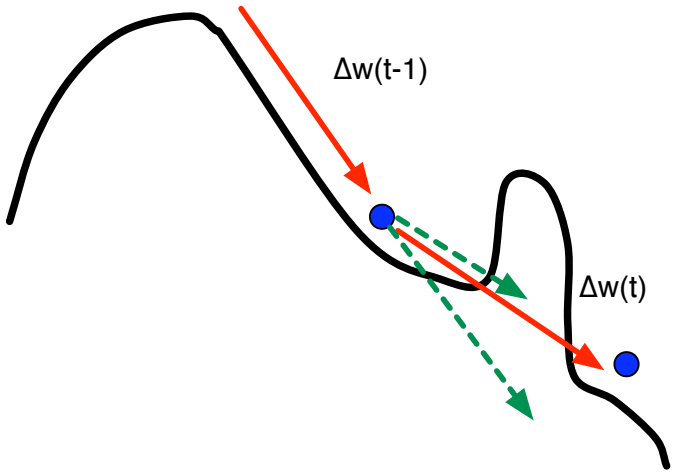

Figure 5: Illustration of the effect of momentum in breaking through local minima. Given the current location of the system on the error landscape (upper blue dot), the standard weight-updating algorithm would move it toward the lowest point of the local minimum, as shown by the upper dotted line. However, the previous move (line labeled $\Delta w(t-1)$) provides momentum (long dotted line). The combination of the two (dotted line) vectors gives the resultant weight change, which breaks through the hump that encloses the local minimum.

To combat the problem, a momentum term is added to the weight-update function such that the current weight change is a combination of the standard change suggested by equation 1 and the change made during the **previous** learning pass of backpropagation, known as the *momentum term*. Equation 39 expresses this update formula, where α is the momentum factor. In many implementations, α decreases with the training epoch such that momentum has a large effect in the early epochs but a minor role later.

$$\Delta w_{ij}(t) = -\eta \frac{\partial E_i}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1) \tag{39}$$

[1]Here, we ignore less conventional, non-monotonic, activation schemes such as radial-basis functions

As shown in Figure 5, this momentum can help the search process blast through the walls that surround a local minimum to continue its descent on the error landscape.

7 Supervised Learning in the Cerebellum

The cerebellum has a well-established role in the learning and control of complex motions [8, 2], and many believe that this involves the use of predictive models [13, 1] refined by a primitive form of supervised learning [5] A brief anatomical overview (based on [2]) of the cerebellum appears in Figure 6, which indicates the highly ordered structure of this region.

As shown in Figures 6 and 7, the cerebellar input layer, the granular cells, receive a variety of peripheral sensory and cortical signals via mossy fibers stemming from the spinal cord and brainstem. These signals experience differential delays before converging upon the granular cells, with an average of 4 such inputs per cell [11] The large number of such cells, approximately 10^{11} in humans [8], combined with their tendency to laterally inhibit one another, via the interspersed golgi cells, indicates that the granular cells serve as sparse-coding detectors of relatively simple (i.e. involving just a few integrated stimuli) contexts [11]. Since delay times vary along the mossy fibers, each context has both temporal and spatial extent.

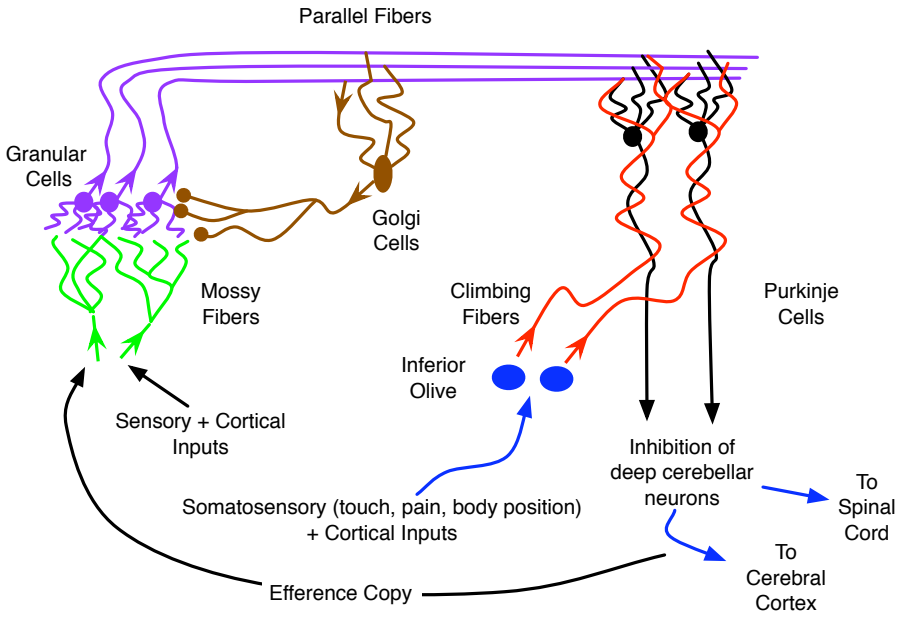

Figure 6: The basic organization of the cerebellum, an abstraction and combination of more complex diagrams in Bear et al. [2], originally appearing in [4].

One parallel fiber emanates from each granular cell and synapses onto the dendrites of many Purkinje cells, each of which may receive input from 10^5 to 10^6 parallel fibers [8]. Since the Purkinje outputs are the cerebellum's ultimate contribution to the control of motor (and possibly cognitive) activity, the plethora granular inputs to each Purkinje cell would appear to embody a complex set of preconditions for the generation of any such output. Since the PF-PC synapses are modifiable [8, 11], these preconditions are subject to learning/adaptation.

As shown in Figure 6, climbing fibers from the inferior olive send signals to the PF-PC synapses. The

climbing fibers transfer pain signals from the muscles and joints controlled by those fibers' corresponding Purkinje cells, and these affect long-term depression (LTD) of the neighboring PF-PC synapses [8, 11]. Thus, the climbing fibers provide a primitive form of supervised learning [5] wherein the combination of parallel fibers that cause a Purkinje cell to fire (and thus promote a muscular movement resulting in discomfort) will be less likely to excite the same PC in the future. In short, the feedback from the inferior olive and clmbing fibers helps to filter out inappropriate contexts (embodied in the parallel fibers) for particular muscle activations.

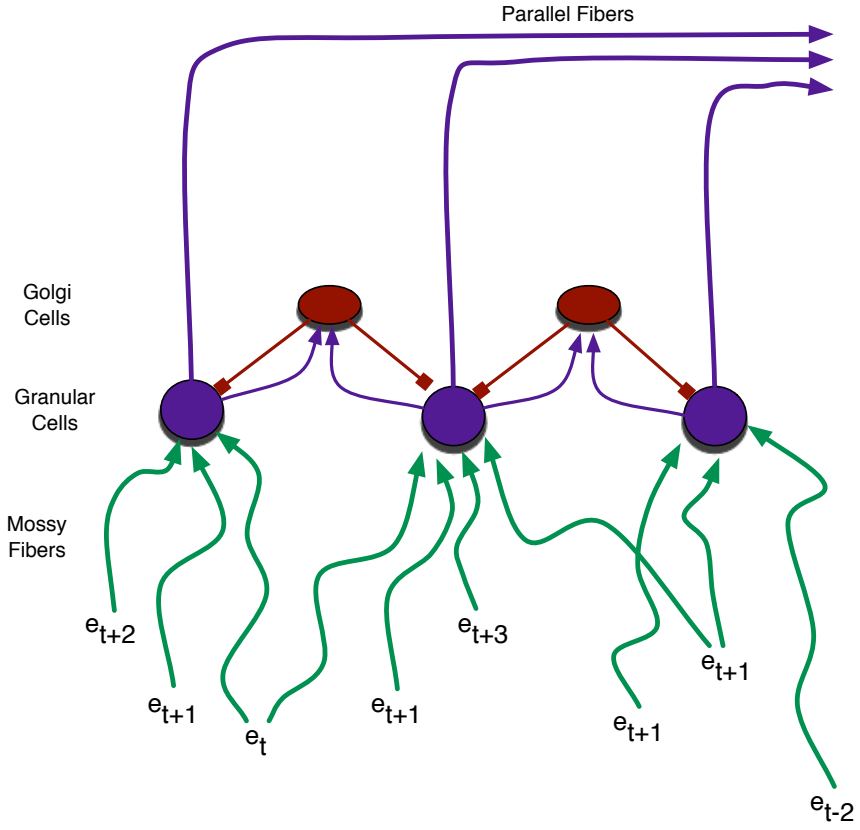

Figure 7: Granular cells realize sparse coding for temporally-blended contexts. Events (e) of various temporal origins (denoted by subscripts) simultaneously activate granular cells due to differential delays - roughly depicted by line length, with longer lines denoting events that occurred further in the past -along mossy fibers.

Plasticity at the PF-PC synapse relies on post-synaptic long-term depression (LTD). When a climbing fiber (CF) forces a PC to fire strongly, those PC dendrites that were recently activated by parallel fibers undergo chemical changes that reduce their sensitivity to glutamate (the neurotransmitter used by PFs). Hence, the influence of those PFs on the PC declines [2].

Somewhat counterintuitively, the simplest behaviors often require the most complex neural activity patterns. For example, it takes a much more intricate combination of excitatory and (particularly) inhibitory signals to wiggle a single finger (or toe) than to move all five. Hence, the tuning of PC cells to achieve the appropriate inhibitory mix is a critical factor in basic skill learning.

Figure 8 gives a hypothetical example of a behavioral rule implemented by a cerebellar tract. A baseball outfielder receives a variety of sensory inputs with different temporal delays, shown here as converging on the

same granular cell. The granular output then affects several Purkinje cells, including those whose ultimate effect is to adjust the player's orientation and leg angle in the attempt to rapidly accelerate toward the projected destination of the ball.

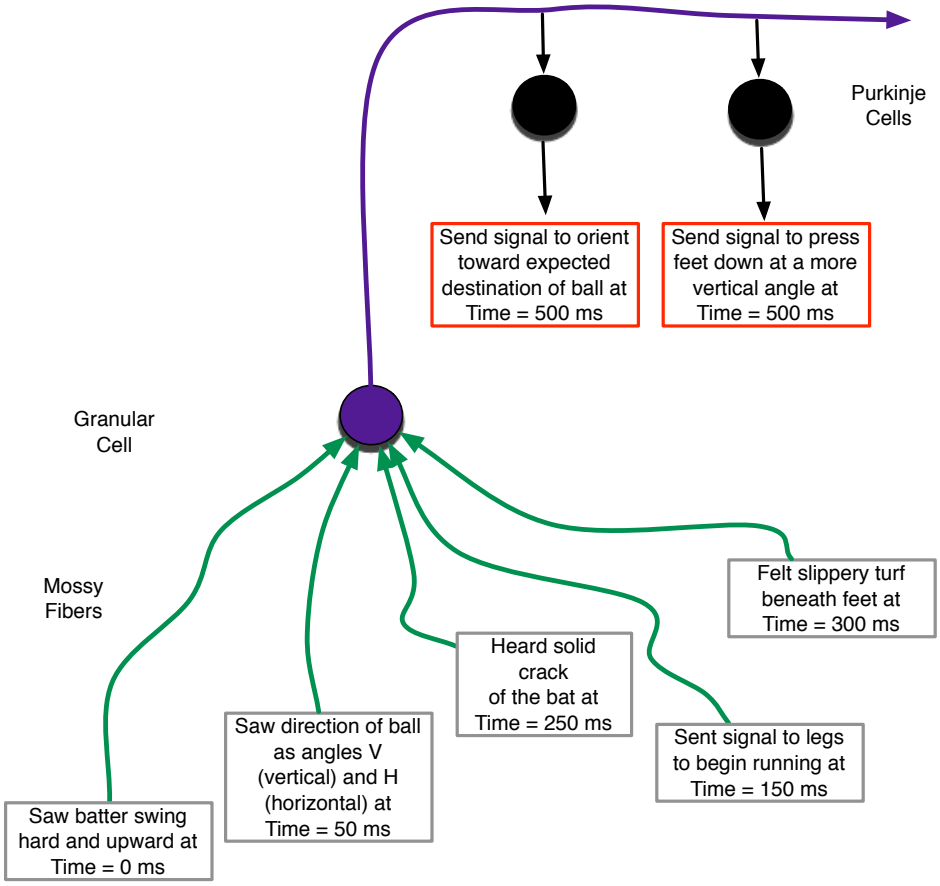

Figure 8: Temporally-mixed sensory and proprioceptive experiences of a baseball outfielder. These form a context for increasing vertical foot plant while accelerating to catch a fly ball.

The *predictiive* nature of this and similar rules involves the integration of sensory stimuli, whose temporal relationships are highly salient, to determine proper actions. Thus, cross-sections of the past determine present decisions about future behaviors. As depicted in Figure 9, the detection of any salient consequences or errors comes even later, due to sensory-processing delays. That error signal should then provide feedback regarding the decisions made earlier.

To maintain an approximate record of what channels were active, and when, and thus what synapses are most *eligible* for modification, the cerebellum and many other brain areas utilize a complex biochemical process that essentially yields a synapse most receptive to LTP or LTD about 100 msec after high transmission activity (as discussed in [9, 7]). This *eligibility trace*, in the parlance of reinforcement learning theory [12], helps compensate for the time delays of sensory processing and motor activation. Eligibility dynamics have probably coevolved with the sensory, motor and proprioceptive apparatus to support optimal learning. Figure 10 shows the eligibility traces associated with several context-action pairs, with those occurring within a narrow time window prior to error detection having the highest values.

Considering that the human cerebellum consists of over a million parallel fibers, each of which embodies

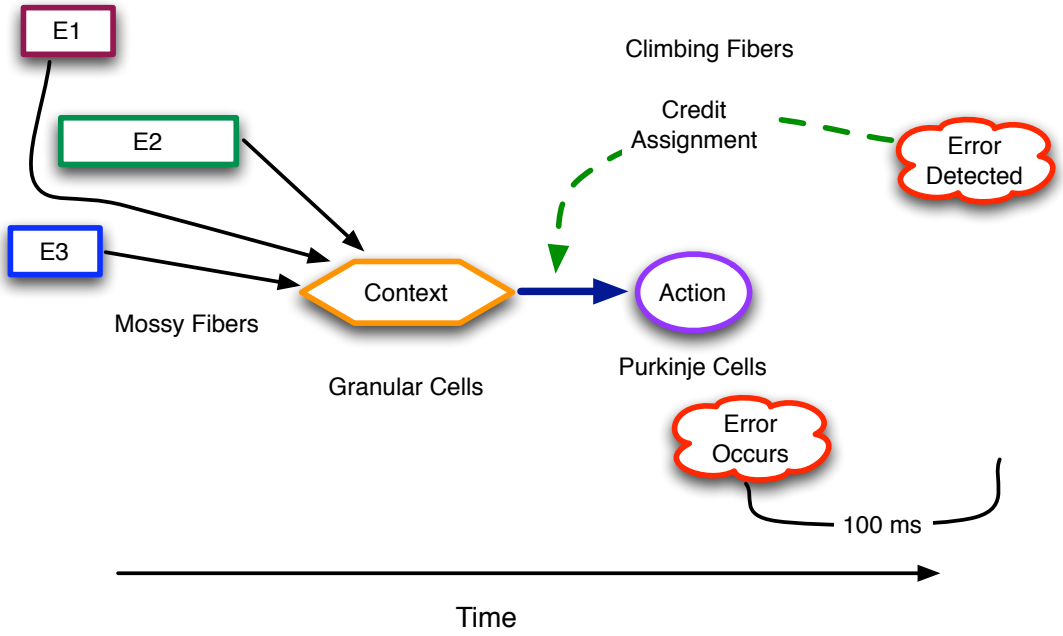

Figure 9: The temporal scope of cerebellar decision making. Context from the past affects present action choices whose actions are realized in the future and whose consequences are perceived even further in the future.

a context-action association, physical skill learning may consist of the gradual tuning and pruning of this immense rule set. Links of high utility should endure, while others will fade via LTD. Importantly, since contexts reflect states of the world prior to action choice and action performance - again, due to inherent sensory-processing delays - the actions that they recommend should be those most appropriate for states of the body and world at some future time (relative to the contexts). Recommendations that lack this predictive nature will produce inferior behavior and be weakened via LTD. By trial and error, the cerebellum learns to support the most salient predictions, which are those that properly account for the inherent delays in sensory processing and motor realization.

From the viewpoint of an outside observer, the cerebellum's actions would appear to involve explicit knowledge as to future states, such as L, the location of the baseball 3 seconds after contact with the bat. However, the cerebellar rule need only embody the behavior that will eventually move the player to that spot, without an explicit representation of the spot itself. For example, in the eye-tracking simulations and primate trials of Kettner et. al., [9], both monkeys and computer models anticipate future points along complex visual trajectories by shifting gaze to the appropriate locations. In describing these systems as predictive, the authors refer to overt behaviors that indicate, to the outside observer, explicit knowledge of future locations. However, neither system is claimed to explicitly house representations (i.e. correlated brain states) for those sites. The predictive knowledge is purely procedural. Knowing how and when to *look* at a location is a lot different than explicitly knowing *about* that spot.

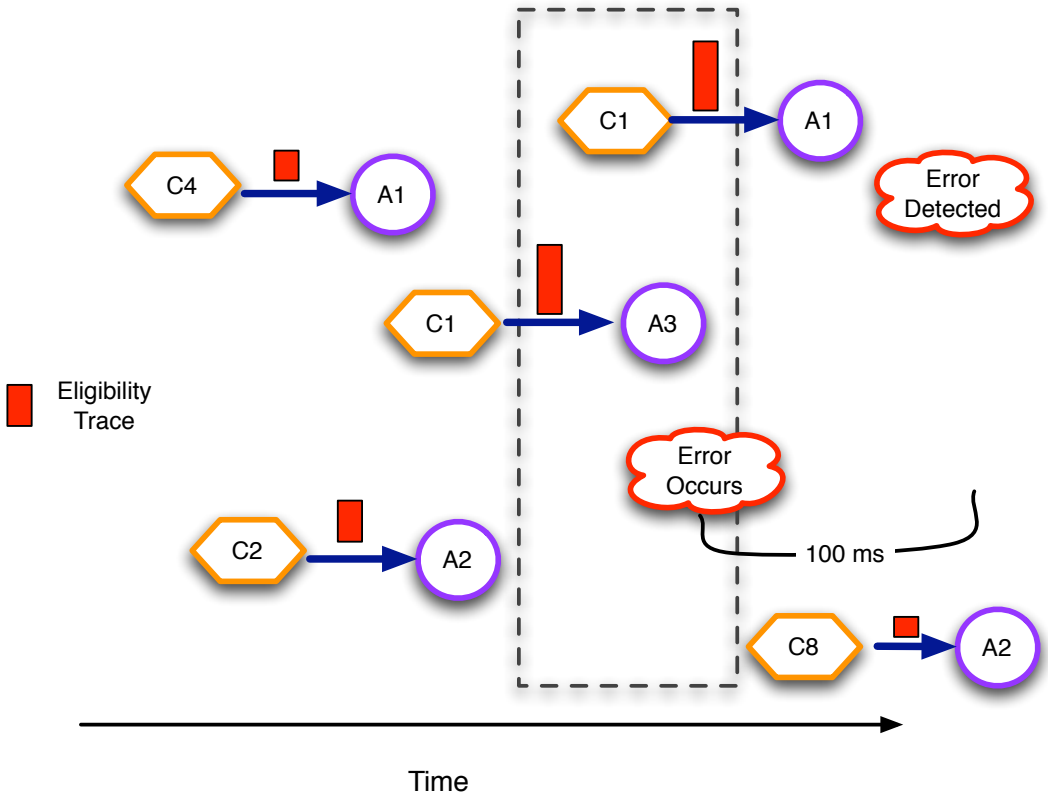

Figure 10: Cerebellar eligibility traces, drawn as rectangles on the condition-action arcs, with taller rectangles denoting higher eligibility. Synapses are most eligible for modification approximately 100 milliseconds after they transmit an action potential.

8 Conclusion

ANNs using supervised learning are a ubiquitous tool in machine learning. Any respectable AI *expert* understands them and their proper domains of application. No ANN course is complete without at least a cursory discussion of the backpropagation algorithm, which is fundamental to the field and a great boon for engineering problem solving (involving the discovery of functional mappings).

However, backpropagation possesses only very weak biological support. A single level of learning (at, for example, the output end of an ANN) based on a reinforcing, error-based signal, has neural plausibility, but the backward transmission of an error signal that gets modified at each level is at odds with conventional neuroscientific wisdom. The cerebellum exhibits a very primitive form of supervised learning in that the feedback (from the inferior olive) comes very often, but it is a far cry from the *correct answer* signal on which backpropagation relies. Still, the view of the cerebellum as a supervised learner helps distinguish its behavior from that of other regions such as the hippocampus (unsupervised learning) and the basal ganglia (reinforcement learning) [5].

Also, the very concept of supervised learning in neural systems serves as a gold standard for tutoring, to which all other neural systems can be juxtaposed and thereby assessed for adaptive potential, since, clearly, any system that receives feedback of a supervisory nature has an exceptional starting point for mastering a given problem domain, whether sensorimotor or more cognitive.

References

[1] N. S. A.G. Barto, A. H. Fagg and J. Houk, *A cerebellar model of timing and prediction in the control of reaching*, Neural Computation, 11 (1999), pp. 565–594.

[2] M. Bear, B. Conners, and M. Paradiso, *Neuroscience: Exploring the Brain*, Lippincott Williams and Wilkins, Baltimore, MD, 2 ed., 2001.

[3] R. Callan, *The Essence of Neural Networks*, Prentice Hall, London, England, 1999.

[4] K. L. Downing, *Neuroscientific implications for situated and embodied artificial intelligence*, Connection Science, 19 (2007), pp. 75–104.

[5] K. Doya, *What are the computations of the cerebellum, the basal ganglia, and the cerebral cortex?*, Neural Networks, 12 (1999), pp. 961–974.

[6] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice Hall, Inc., Upper Saddle River, N.J., 1999.

[7] J. Houk, J. Adams, and A. Barto, *A model of how the basal ganglia generate and use neural signals that predict reinforcement*, in Models of Information Processing in the Basal Ganglia, J. Houk, J. Davis, and D. Beiser, eds., Cambridge, MA, 1995, The MIT Press, pp. 249–270.

[8] E. Kandel, J. Schwartz, and T. Jessell, *Principles of Neural Science*, McGraw-Hill, New York, NY, 2000.

[9] R. Kettner, S. Mahamud, H. Leung, N. Sitkoff, J. Houk, and B. Peterson, *Prediction of complex two-dimensional trajectories by a cerebellar model of smooth pursuit eye movement*, Journal of Neurophysiology, 77 (1997), pp. 2115–2130.

[10] T. Mitchell, *Machine Learning*, WCB/McGraw-Hill, Boston, MA, 1997.

[11] E. Rolls and A. Treves, *Neural Networks and Brain Function*, Oxford University Press, New York, 1998.

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

[13] D. Wolpert, R. C. Miall, and M. Kawato, *Internal models in the cerebellum*, Trends in Cognitive Sciences, 2 (1998), pp. 338–347.