Reinforcement Learning and Associative Conditioning

(Chapter 7)
The error gradient component w.r.t. any weight can be estimated thus:

Make a small change $\delta w_1$ in $w_1$. Let $\delta w_1$ (bold) be the corresponding change as a matrix (all other entries 0).

Measure the attendant change in error $E$:

$$\frac{\delta E}{\delta w_1} = \frac{E(w + \delta w_1) - E(w)}{\delta w_1}$$

Estimate $\partial E/\partial w_1$ by $\delta E/\delta w_1$. 
Repeat:
   Pick random weight variable.
   Perturb, as in previous slide.
   Use gradient estimate for descent.
Perturbation in a 2-layer, 1-output network

% pertGradientOneByOne.m
% trains two-layered pattern associating networks (with only one output unit) using single weight perturbation to estimate
% the error gradient; initial weight matrix V and input InPat and desired output DesOut patterns must be supplied in the worksp

[nPat,nIn]=size(InPat); % determine number of input units
[nPat,nOut]=size(DesOut); % determine number of output units
Out=zeros(nPat,nOut); % set up output hold array
a=0.01; % set learning rate
pSize=0.005; % set perturbation size
tol=0.1; % set error tolerance within which training is adequate
count=0; % set the iteration counter to zero
countLimit=100000; % set count limit over which training stops

Q=V*InPat'; % find the weighted input sums for all patterns
Out=1./(1+exp(-Q)); % squash to find the output for all patterns
error=sum(abs(DesOut-Out')); % find initial error over all patterns
while error>tol, % while actual error is over tolerance
    pert=pSize*sign(randn); % generate a single weight perturbation
    vChoose=ceil(nIn*rand); % choose one weight from V at random
    V(vChoose)=V(vChoose)+pert; % perturb this weight
    Q=V*InPat'; % find the weighted input sum for all the patterns
    Out=1./(1+exp(-Q)); % squash to find the output for all patterns
    newErr=sum(abs(DesOut-Out')); % find new error over all patterns
    delErr=newErr-error; % find change in error due to perturbation
    estGrad=delErr/pert; % compute estimated gradient
    vDelta=-a*estGrad; % compute weight change
    V(vChoose)=V(vChoose)+vDelta; % apply weight change to weight
    error=newErr; % save new error as the error
    count=count+1; % increment the counter
    if count>countLimit, break, end % break if counter over limit
end
Q=V*InPat'; % find the weighted input sums for all patterns
Out=1./(1+exp(-Q)); % squash to find the output for all patterns
### Table 7.1

Learning the labeled-line patterns using gradient estimation at one weight at a time

<table>
<thead>
<tr>
<th>Input patterns</th>
<th>Desired output patterns</th>
<th>Actual output patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1</td>
<td>0.97</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0</td>
<td>0.02</td>
</tr>
</tbody>
</table>

The input, desired output, and actual output following training are listed for a two-layered, feedforward network of sigmoidal units. The network has four input units and one output unit (a 4-by-1 network). Note that the labeled-line input patterns are non-overlapping.

*TUTORIAL ON NEURAL SYSTEMS MODELING*, Table 7.1

<table>
<thead>
<tr>
<th>To output unit</th>
<th>From input unit</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>+3.71</td>
<td>+3.62</td>
<td>−3.42</td>
</tr>
</tbody>
</table>

TABLE 7.2  Weight matrix after training on the labeled-line (non-overlapping) patterns using gradient estimation at one weight at a time
The previous perturbation method is slow, as only one weight is adjusted per cycle.

We desire to speed this up by perturbing and adjusting multiple weights in each cycle. The following paper, from NIPS, 1993, showed how:

---

A Parallel Gradient Descent Method for Learning in Analog VLSI Neural Networks

---

J. Alspector  R. Meir*  B. Yuhas  A. Jayakumar  D. Lippe†
Bellcore
Morristown, NJ 07962-1910
Use a uniform magnitude $|\delta w_1|$ among all weights, but random signs:

$$\frac{\delta E}{\delta w_1} = \frac{E(w + \delta w) - E(w)}{\delta w_1}$$

which by a Taylor expansion is

$$\frac{\delta E}{\delta w_1} = \sum_{i=1}^{W} \frac{\partial E}{\partial w_i} \frac{\delta w_i}{\delta w_1} + O([\delta w_1])$$

leading to the approximation (ignoring higher order terms)

$$\frac{\delta E}{\delta w_1} = \frac{\partial E}{\partial w_1} + \sum_{i>1}^{W} \left( \frac{\partial E}{\partial w_i} \right) \left( \frac{\delta w_i}{\delta w_1} \right)$$
\[ \frac{\delta E}{\delta w_1} = \frac{\partial E}{\partial w_1} + \sum_{i>1}^{W} \left( \frac{\partial E}{\partial w_i} \right) \left( \frac{\delta w_i}{\delta w_1} \right). \]  

(5)

An important point of this paper, emphasized by (Dembo, 1990) and embodied in Eq. (5), is that the last term has expectation value zero for random and independently distributed $\delta w_i$ since the last expression in parentheses is equally likely to be +1 as -1. Thus, one can approximately follow the gradient by perturbing all weights at the same time. If each synapse has access to information about the resulting change in error, it can adjust its weight by assuming it was the only weight perturbed. The weight change rule

\[ \Delta w_i = -\eta \frac{\delta E}{\delta w_i}, \]  

(6)

where $\eta$ is a learning rate, will follow the gradient on the average but with the considerable noise implied by the second term in Eq. (5).

reminiscent of Brownian motion where, although particles may be subject to considerable random motion, there is a general drift of the ensemble of particles in the direction of even a weak external force. In this respect, there is some similarity to the directed drift algorithm of (Venkatesh, 1991), although that work applies to binary weights and single layer perceptrons whereas this algorithm should work for any level of weight quantization or precision - an important advantage for VLSI implementations - as well as any number of layers and even for recurrent networks.
2.2 Improving the Estimate by Multiple Perturbations

As was pointed out by (Dembo, 1990), for each pattern, one can reduce the variance of the noise term in Eq. (5) by repeating the random parallel perturbation many times to improve the statistical estimate. If we average over \( P \) perturbations, we have

\[
\frac{\delta E}{\delta w_1} = \frac{1}{P} \sum_{p=1}^{P} \frac{\delta E}{\delta w_1^p} = \frac{\partial E}{\partial w_1} + \frac{1}{P} \sum_{p=1}^{P} \sum_{i>1}^{W} \left( \frac{\partial E}{\partial w_i} \right) \left( \frac{\delta w_1^p}{\delta w_1} \right)
\]

(7)

where \( p \) indexes the perturbation number. The variance of the second term, which is a noise, \( \nu \), is

\[
< \nu^2 > = \frac{1}{P^2} \sum_{p,p'=1}^{P} \sum_{i>i'}^{W} \left( \frac{\partial E}{\partial w_i} \right) \left( \frac{\partial E}{\partial w_{i'}} \right) \left( \frac{\delta w_1^p}{\delta w_1} \right) \left( \frac{\delta w_{1'}^{p'}}{\delta w_1} \right)
\]

(8)

where the expectation value, \(< >\), leads to the Kronecker delta function, \( \delta_{ii'}^{pp'} \). This reduces Eq. (8) to

\[
< \nu^2 > = \frac{1}{P^2} \sum_{p=1}^{P} \sum_{i>1}^{W} \left( \frac{\partial E}{\partial w_i} \right)^2.
\]

(9)

The double sum over perturbations and weights (assuming the gradient is bounded and all gradient directions have the same order of magnitude) has magnitude \( O(PW) \) so that the variance is \( O(\frac{W}{P}) \) and the standard deviation is

\[
< \nu^2 >^{\frac{1}{2}} = O \left( \sqrt{\frac{W}{P}} \right).
\]

(10)
Figure 4. Diagram of perturbative learning synapse.
% pertGradientParallel.m
% this script trains two-layered pattern associating networks using parallel weight perturbations to estimate the error gradient;
% the initial weight matrix V and input InPat and desired output DesOut patterns must be supplied in the workspace

[nPat,nIn]=size(InPat); % determine number of input units
[nPat,nOut]=size(DesOut); % determine number of output units
Out=zeros(nPat,nOut); % set up output hold array
a=0.01; % set learning rate
pSize=0.005; % set perturbation size
tol=0.1; % set error tolerance within which training is adequate
count=0; % set the iteration counter to zero
countLimit=100000; % set count limit over which training stops

Q=V*InPat'; % find the weighted input sums for all patterns
Out=1./(1+exp(-Q)); % squash to find the output for all patterns
error=sum(sum(abs(DesOut-Out'))); % initial error over all patterns

while error>tol,
    Pert=pSize*sign(randn(nOut,nIn)); % parallel weight perturbation
    V=V+Pert; % apply perturbation of all weights in parallel
    Q=V*InPat'; % find the weighted input sum for all the patterns
    Out=1./(1+exp(-Q)); % squash to find the output for all patterns
    newErr=sum(sum(abs(DesOut-Out'))); % new error over all patterns
    delErr=newErr-error; % find change in error due to perturbation
    estGrad=delErr./Pert; % compute matrix of estimated gradients
    deltaV=-a*estGrad; % compute matrix of weight changes
    V=V+deltaV; % apply weight change matrix to weight matrix
    error=newErr; % save new error as the error
    count=count+1; % increment the counter
    if count>countLimit, break, end % break if counter over limit
end

Q=V*InPat'; % find the weighted input sums for all patterns
Out=1./(1+exp(-Q)); % squash to find the output for all patterns
### TABLE 7.3
Learning the labeled-line patterns using gradient estimation at all weights in parallel

<table>
<thead>
<tr>
<th>Input patterns</th>
<th>Desired output patterns</th>
<th>Actual output patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0</td>
<td>0.04</td>
</tr>
</tbody>
</table>

The input, desired output, and actual output following training are listed for a two-layered, 4-by-1 feedforward network of sigmoidal units. Note that the labeled-line input patterns are non-overlapping.

### TABLE 7.4
Weight matrix after training on the labeled-line (non-overlapping) patterns using gradient estimation at all weights in parallel

<table>
<thead>
<tr>
<th>To output unit</th>
<th>From input unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>+4.04</td>
</tr>
</tbody>
</table>
TABLE 7.5  Failure to learn the complex contingency patterns using gradient estimation at all weights in parallel

<table>
<thead>
<tr>
<th>Input patterns</th>
<th>Desired output patterns</th>
<th>Actual output patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>0</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The input, desired output, and actual output following training are listed for a two-layered, 4-by-1 feedforward network of sigmoidal units. Note that the

TABLE 7.6  Weight matrix after unsuccessful training on the complex contingency (overlapping) patterns using gradient estimation at all weights in parallel

<table>
<thead>
<tr>
<th>To output unit</th>
<th>From input unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>1</td>
<td>-2.36 +3.99 +9.64 -4.59</td>
</tr>
</tbody>
</table>
TABLE 7.7  Successfully learning the complex contingency patterns using gradient estimation at all weights in parallel

<table>
<thead>
<tr>
<th>Input patterns</th>
<th>Desired output patterns</th>
<th>Actual output patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1</td>
<td>0.97</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0</td>
<td>0.04</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>0</td>
<td>0.02</td>
</tr>
</tbody>
</table>

The input, desired output, and actual output following training are listed for a two-layered, 4-by-1 feedforward network of sigmoidal units. Training began from an initial random state different from that which generated the unsuccessful learning results in Table 7.5.

TABLE 7.8  Weight matrix after successful training on the complex contingency (overlapping) patterns using gradient estimation at all weights in parallel

<table>
<thead>
<tr>
<th>To output unit</th>
<th>From input unit</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>-10.69</td>
<td>-7.45</td>
<td>+14.84</td>
<td>+6.86</td>
</tr>
</tbody>
</table>

Training began from an initial, random state different from that which generated the unsuccessful learning results in Table 7.5. The weights in this table are different from those in Table 7.6.
The author mentions that perturbation method is more dependent on weight initialization than backpropagation.

At the same time, it is more biologically feasible.
Adding Reinforcement to Perturbation
**Directed Drift** is an improvement using negative reinforcement learning:

Weights are perturbed for a trial, but accepted only if performance increases.

Otherwise the previous weights are reinstalled.
% pertDirectedDrift.m
% this script trains two-layered pattern associating networks using parallel weight perturbation via the directed drift algorithm;
% the initial weight matrix V and input InPat and desired output DesOut patterns must be supplied in the workspace

[nPat,nLn]=size(InPat); % determine number of input units
[nPat,nOut]=size(DesOut); % determine number of output units
Out=zeros(nPat,nOut); % set up output hold array
a=0.01; % set learning rate
tol=0.1; % set error tolerance within which training is adequate
count=0; % set the iteration counter to zero
countLimit=100000; % set count limit over which training stops

Q=V*InPat'; % find the weighted input sums for all patterns
Out=1./(1+exp(-Q)); % squash to find the output for all patterns
error=sum(sum(abs(DesOut-Out')))); % initial error over all patterns

while error>tol,
    Pert=a*sign(randn(nOut,nLn)); % weight perturbation matrix
    % Pert=a*randn(nOut,nLn); % alternate weight perturbation matrix
    holdV=V; % hold the current weight matrix
    V=V+Pert; % apply perturbation of all weights in parallel
    Q=V*InPat'; % find the weighted input sum for all the patterns
    Out=1./(1+exp(-Q)); % squash to find the output for all patterns
    newErr=sum(sum(abs(DesOut-Out')))); % new error over all patterns
    if newErr>=error, % if the perturbation increases the error
        V=holdV; % then remove the weight perturbation matrix
    elseif newErr<error, % leave perturbation if it decreased error
        error=newErr; % save new error as the error
    end
    count=count+1; % end conditional
    if count>countLimit, break, end % break if counter over limit
end

Q=V*InPat'; % find the weighted input sums for all patterns
Out=1./(1+exp(-Q)); % squash to find the output for all patterns
TABLE 7.9 Successfully learning the complex contingency patterns using parallel, pertubative reinforcement learning

<table>
<thead>
<tr>
<th>Input patterns</th>
<th>Desired output patterns</th>
<th>Actual output patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>1</td>
<td>0.97</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0</td>
<td>0.03</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>0</td>
<td>0.02</td>
</tr>
</tbody>
</table>

The input, desired output, and actual output following training are listed for a two-layered, 4-by-1 feedforward network of sigmoidal units. On each training cycle, the parallel weight perturbation was followed by retention of the perturbation or restoration of the unperturbed weights if reinforcement was positive or negative, respectively. The initial weight matrix was $V = [-1 -1 +1 +1]$. 
**Table 7.10**  Weight matrix after successful training on the complex contingency (overlapping) patterns using parallel, perturbative reinforcement learning

<table>
<thead>
<tr>
<th>To output unit</th>
<th>From input unit</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>−10.80</td>
<td>−7.32</td>
<td>+14.76</td>
<td>+6.96</td>
</tr>
</tbody>
</table>

The initial weight matrix was $V = [-1 -1 +1 +1]$.  

*Tutorial on Neural Systems Modeling, Table 7.10*  
Perturbation methods can also be applied to networks with more than 2 layers.

Such networks are capable of learning distributed representations, similar to what we saw with backpropagation.

However, with perturbation, the distributed nature tends to be even more diverse.
### Table 7.11 Applying Perturbation to 3-layer Feedforward Network

#### TABLE 7.11  
**Learning the simple labeled-line patterns using parallel, perturbative reinforcement**

<table>
<thead>
<tr>
<th>Input patterns</th>
<th>Desired output patterns</th>
<th>Actual output patterns</th>
<th>Hidden unit responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>0.06</td>
<td>0.08</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>0.98</td>
<td>0.91</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>0.98</td>
<td>0.90</td>
</tr>
</tbody>
</table>

The input, desired output, actual output, and the responses of the hidden unit following training are listed for a three-layered, feedforward network of sigmoidal units. The network has two input, one hidden, and one output unit.
**TABLE 7.12**  Weight matrices after perturbative reinforcement training on the simple labeled-line patterns

<table>
<thead>
<tr>
<th>To hidden unit</th>
<th>From input unit</th>
<th>From bias unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>+4.72</td>
<td>+4.62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>To output unit</th>
<th>From hidden unit</th>
<th>From bias unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>+8.11</td>
<td>−3.33</td>
</tr>
</tbody>
</table>
Figure 7.4  Simulating a non-uniform distributed representation
Figures 7.4 and 6.6 Compared: Simulating a non-uniform distributed representation

Backpropagation

Perturbation
In classical conditioning, the subject learns to produce an involuntary response to a stimulus to which it would not ordinarily respond.

Example: Pavlov’s dog experiment
Pavlov's Dogs

Pavlov used a bell to call to their food and, after a few repetitions, the dogs started to salivate in response to the bell.

Pavlov called the bell the **conditioned stimulus (CS)** because its effect depended on its association with food.

He called the food the **unconditioned stimulus (US)** because its effect did not depend on previous experience.

Likewise, the response to the CS was the **conditioned response (CR)** and the response to the US was the **unconditioned response (UR)**.

The **timing** between the presentation of the CS and US is integral to facilitating the conditioned response. Pavlov found that the shorter the interval between the bell's ring and the appearance of the food, the more quickly the dog learned the conditioned response and the stronger it was.

In operant conditioning, the subject learns to produce an voluntary response to a stimulus due to reinforcement.

Example: Skinner’s rat experiment
An operant conditioning chamber permits experimenters to study behavior conditioning (training) by teaching a subject animal to perform certain actions (like pressing a lever) in response to specific stimuli, like a light or sound signal. When the subject correctly performs the behavior, the chamber mechanism delivers food or another reward. In some cases, the mechanism delivers a punishment for incorrect or missing responses. With this apparatus, experimenters perform studies in conditioning and training through reward/punishment mechanisms.

Some types of conditioning combine both forms.

Example:

Air puff in rabbit’s eye produces eye blink.
Tone does not.

After presentation of tone along with air puff, rabbit learns to associate the two.

Then tone by itself will produce eye blink.
Association must occur here.
A rabbit is caged with a freely-spinnable wheel. The floor can be electrified.

Training:
When a tone is sounded, the rabbit is given a brief shock, unless the rabbit, by exploration, happens to spin the wheel, which will prevent the shock.

The rabbit learns that the tone is followed by shock, and learns to spin the wheel in response to the tone.
Avoidance Conditioning Schema (Higher-Level Neuronal) Explanation

- **TONE:** The tone that is sounded
- **HEAR:** The rabbit hearing the tone
- **SUMO:** The rabbit spinning the wheel
- **FUMO:** The rabbit “freezing” (not spinning the wheel)
Figure 7.6 Multiunit responses of neurons in the medial geniculate nucleus and the anterior and posterior cingulate cortex during avoidance conditioning in rabbits.
Figure 7.7  Simple schema-theoretic model of avoidance conditioning

- Excitatory
- Modifiable
- Modulatory
- Interactive

TONE → HEAR → SUMO → FUMO

 schema

Reward for spin

Reward for freeze

- e.g. +1
- e.g. 0

TUTORIAL ON NEURAL SYSTEMS MODELING, Figure 7.7
Figure 7.8  Simulation of avoidance conditioning
Figure 7.9 Simulation of avoidance conditioning at a higher probability of exploration
The CALL schema can increase probability of exploration based on the PAIN schema.

SPIN inhibits SHOCK
SHOCK activates PAIN
PAIN modulates CALL
CALL activates exploration

Observation: As the rabbit learns to avoid shock, the CALL is gradually activated less.
Figure 7.10 Schema theoretic model of avoidance conditioning in which the probability of exploration can be augmented.
% avoidanceLearnCall.m
% this script models avoidance learning as a reinforcement learning
% with two upper motoneurons (sumo and fumo) and one "call" neuron

nTrials=2000; % set number of learning trials
a=0.005; % set learning rate
bprob=0.005; % set baseline probability of exploration
wsh=0; % set initial modifiable weight to sumo from hear
wfh=1; % set initial modifiable weight to fumo from hear
wch=0; % set initial modifiable weight to call from hear
hear=1; % set response of hear

for c=1:nTrials,
    if c<=nTrials/10,
        rews=0; rewf=1;
    else
        rews=1; rewf=0;
    end
    call=wch*hear;
    sumo=wsh*hear;
    fumo=wfh*hear;
    spin=sumo>fumo;
    prob=bprob+bprob*call;
    if prob>rand, spin=1-spin; end % explore sometimes
    callrec(c)=call;
    sumorec(c)=sumo;
    fumorec(c)=fumo;
    spinrec(c)=spin;
    if c<=nTrials/10,
        pain=0; % no pain is delivered
    elseif c>nTrials/10 & spin==1,
        pain=0; % pain is avoided
    elseif c>nTrials/10 & spin==0,
        pain=1; % pain is not avoided
    end % end pain conditional
    wch=wch+a*(pain-wch);
    if spin==1,
        wsh=wsh+a*(rews-wsh);
    else
        wfh=wfh+a*(rewf-wfh);
    end % end update conditional
end % end main training loop
Figure 7.11 Simulation of avoidance conditioning with an adjustable probability of exploration