**Level 5: AI Game Programming**

Lab 5

**Simple multi-layer ANN and backpropagation**

RECORD ALL THAT YOU DO IN YOUR LAB LOGBOOK (HARDCOPY ‘OR’ WORD FILE)

**Continue from Lab 4**

**Backpropagation**

**Simple multi-layer ANN and backpropagation**

**Training a ‘simple’ multilayer neural network to learn a simple task.**

On some occasions, the learning strategy is for you to have some experience in figuring out what is going on in the lab, and then I'll reinforce it with a detailed explanation in the following lecture. It's more effective this way to 'train' your brain to master the subject.

Actually, this is similar to many training methods used for artificial neuron networks. The networks are incrementally adjusted until they can 'learn' the subject or task.

Therefore, if you do not quite understand all the details in the lab, that's alright. You only need to 'try' it. This will prepare your brain for the detailed discussion in the following lecture. (For those with a strong background, you can treat it as an exercise to refresh your memory.)

Note that training an ANN takes time; if you can't finish everything in the lab, you can carry on training and try it on your own time or in the next lab.

**Backpropagation**

Consider the summary of the single-neuron training:

Diagram

Description automatically generated

At the end of the lecture on Monday, I asked:

Diagram

Description automatically generated with medium confidenceDiagram

Description automatically generated

To solve the problem, we can propagate the error back to other neurons.

A picture containing chart

Description automatically generated

The error is propagated back ***through the relevant weights*** to the relevant neurons.

Radar chart

Description automatically generated with low confidenceSchematic

Description automatically generated

This is the most common type of neural networks where the data or signal flows forward and the error is propagated back to adjust the weights.

Diagram

Description automatically generated

**Error and error gradient**

In this lab, Sigmoid function will be used as the activation function and 'error gradient ' will be used instead of 'error (difference between the current output and the correct output), see the figure below. Why? Please think about it, again to prepare you brain, before we discuss it in detail in the following lecture.

A diagram of a mathematical equation

Description automatically generated

**TASK**

Write Python code using Google Colab to implement the simple multilayer neural network above and the backpropagation training to make the network learn to handle a non-player character’s decision making process in your game - that is, whether the creature will attack of flee, depending on the creature’s power and enemy’s power.

You may notice that the number of inputs and outputs in this example are still the same as those in our single-neuron case. This is because it's easier to manage and study. If this works, then you can easier the same method to more inputs, outputs and neurons.

(Note that for this type of decision, one neuron is not enough. If you wish, you could try it.)

|  |  |  |
| --- | --- | --- |
| INPUT | | OUTPUT |
| Creature’s power | Enemy’s power | Creature’s decision/action |
| Strong (1) | Strong (1) | Attack (0) |
| Weak (0) | Strong (1) | Flee (1) |
| Strong (1) | Weak (0) | Flee (1) |
| Weak (0) | Weak (0) | Attack (0) |

*- See an example pseudocode below; there are gaps you still need to fill. Then implement the code in Python using Google Colab.*

*- Pay attention to the epoch sum error used to measure if the training converts to the correct weights and biases.*

*- Check your results against the table. Does the training work?*

**Example pseudocode for this training**

NB: This is an example of a very simple implementation. Once your code works, you can improve it to make it more effective and expandable to cope with more neurons, e.g. use functions and objects to represent each neuron, etc.

% AIGP Labs 5

Niteration = 1000000; % Number of iterations

%(by this number, you should have the result,

% otherwise, there may be something wrong in your code.

% Set initial weights and bias (normally, they are random numbers, but to

% make debugging easier, we begin with a fixed set of weights and biases

w(1,3) = 0.5; w(2,3) = 0.4;

w(1,4) = 0.9; w(2,4) = 1.0;

w(3,5) = -1.2; w(4,5) = 1.1;

Bias(3) = 0.8; Bias(4) = -0.1; Bias(5) = 0.3;

Alpha = 0.1; % Learning rate

%Prepare training data - inputs and correct outputs according to the

%decision table for the ANN to learn

x1(1) = 1; x2(1) = 1; Yd5(1) = 0; % Case 1

x1(2) = 0; x2(2) = 1; Yd5(2) = 1; % Case 2

x1(3) = 1; x2(3) = 0; Yd5(3) = 1; % Case 3

x1(4) = 0; x2(4) = 0; Yd5(4) = 0; % Case 4

p = 1; % iteration number

while p <= Niteration

EpocSumError = 0; % Initiate the termination condition value

% Begining of an Epoch

for i = 1:4 % Case number

p % print iteration number

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% FEED FORWARD

% Calculate output for neuron 3

X(3) = x1(i)\*w(1,3)+x2(i)\*w(2,3)+Bias(3);

% Activation Function - Sigmoid function ()

% Note: exp(1) = e^1 = 2.7183 (Euler’s number)  
 % (see the labsheet and lecture notes last week)

y(3) = …..[You need to find out here how to calculate output y(3) of neuron 3]

% Calculate output for neuron 4

X(4) = x1(i)\*w(1,4)+x2(i)\*w(2,4)+Bias(4);

% Activation Function - Sigmoid function

y(4) = …..[You need to find out here how to calculate output y(4) of neuron 3]

% Calculate output for neuron 5

X(5) = y(3)\*w(3,5)+y(4)\*w(4,5)+Bias(5);

% Activation Function - Sigmoid function

y(5) = …..[You need to find out here how to calculate output y(5) of neuron 3]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% BACK PROPAGATION

% Back propagate the error and use the learning rule to update the

% weights

% Starting from neuron 5

%........................................................

% COST FUNCTION

% Calculate the cost (error) at the output of the network to propage back

% through the network

% Note that this one is the simplest cost (error) function (see the

% lecture notes). It is used to update the weights in the case,

% then use the new weights for the next case in the Epoch.

% Later, you could try other cost function such as Quadratic cost

% (see the lecture notes), but be aware, for Quadratic cost, you

% first calculate the cost (error) from all cases in the Epoch, then use

% that cost to propagate back and update the weights.

e(5) = Yd5(i) - y(5);

%..................................................................

Delta(5) = y(5)\*(1-y(5))\*e(5);

Wcurrent(3,5) = w(3,5); % safe the current value before updating it

Wcurrent(4,5) = w(4,5); % safe the current value before updating it

w(3,5) = w(3,5) + Alpha\*y(3)\*Delta(5);

w(4,5) = w(4,5) + Alpha\*y(4)\*Delta(5);

Bias(5) = Bias(5) + Alpha\*(1)\*Delta(5);

% Neuron 3

Delta(3) = y(3)\*(1-y(3))\*Delta(5)\*Wcurrent(3,5);

w(1,3) = w(1,3) + Alpha\*x1(i)\*Delta(3);

w(2,3) = w(2,3) + Alpha\*x2(i)\*Delta(3);

Bias(3) = Bias(3) + Alpha\*(1)\*Delta(3);

% Neuron 4

Delta(4) = y(4)\*(1-y(4))\*Delta(5)\*Wcurrent(4,5);

w(1,4) = w(1,4) + Alpha\*x1(i)\*Delta(4);

w(2,4) = w(2,4) + Alpha\*x2(i)\*Delta(4);

Bias(4) = Bias(4) + Alpha\*(1)\*Delta(4);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Check the error for the whole epoch (all cases) to see if the set

% of the weights and biases produce correct outputs for all cases

% (Epoch).

% Note that if you use Quadratic cost, then you can use the

% Quadratic cost value as the condition, don't need to calculate

% this part

% Accumulating the sum of 'squared' errors in the Epoch

% Recalculate the error at the end e(5) after updating all the

% weights and biases

% [Check what the code below means, why?]

tX(3) = x1(i)\*w(1,3)+x2(i)\*w(2,3)+Bias(3);

ty(3) = 1/( 1 + (exp(1))^(-(tX(3))));

tX(4) = x1(i)\*w(1,4)+x2(i)\*w(2,4)+Bias(4);

ty(4) = 1/( 1 + (exp(1))^(-(tX(4))));

tX(5) = ty(3)\*w(3,5)+ty(4)\*w(4,5)+Bias(5);

ty(5) = 1/( 1 + (exp(1))^(-(tX(5))));

te(5) = Yd5(i) - ty(5);

% Squared error

EpocSumError = EpocSumError + (te(5))^2; %[What is EpocSumError? What does it do?]

p = p+1;

end % for i = 1:4

% The training process is repeated until the sum of squared error in

% The Epoch is acceptable, i.e. less than, for example, 0.001. We have to do this because you won’t get the exact zero error

\*\*\* Put termination condition here \*\*\*\*

EpocSumError

if EpocSumError < 0.001 % Example

break

end

end % while p <= Niteration

***When programming, it is a good practice to check carefully. For each step, check the results against the equations to ensure that the code is correct before running the whole thing for many iterations. This is a good way to practice implementing and debugging your code.***

For example, in the first iteration, the values of the variables should be as follows. If your code produces results far from the values below, then you need to check for bugs:  
  
First iteration p = 1  
y(3) = 0.8455  
y(4) = 0.8581  
y(5) = 0.5571  
e(5) = -0.5571  
Delta(5) = -0.1375  
w(3,5) = -1.2116  
w(4,5) = 1.0882  
Bias(5) = 0.2863  
Delta(3) = 0.0215  
w(1,3) = 0.5022  
w(2,3) = 0.4022  
Bias(3) = 0.8022  
Delta(4) = -0.0184  
w(1,4) = 0.8982  
w(2,4) = 0.9982  
Bias(4) = -0.1018  
te(5) = -0.5483  
EpocSumError = 0.3007

**SOLUTIONS**

The final weights and biases should be (approximately):

w(1,3) = 4.5742  
w(1,4) = 6.6165  
w(2,3) = 4.5669  
w(2,4) = 6.5849  
w(3,5) = -10.5474  
w(4,5) = 9.7713

Bias(3) = -7.0188; Bias(4) = -2.9530; Bias(5) = -4.5013

Note that, from a multi-layer ANN, you won’t get the clear, exact values for the desired outputs and exact zero errors. For example:

A screenshot of a graph

Description automatically generated

**END**