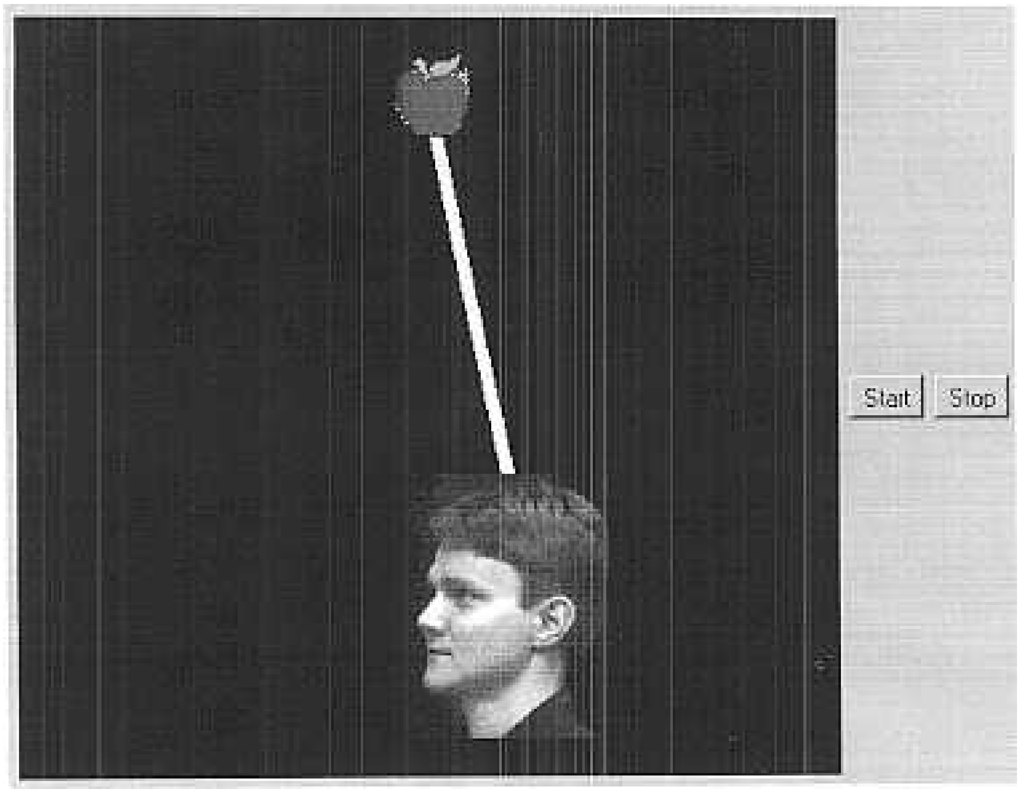
# Pole Balancing: A Neural Network Approach



Daniel Mess
Cornell University
May 17,1999

***http*** : / */www.people.* ***cornell.*** *edu/pages/* ***dah2*** *3/pole/pole.* ***html***

0-

## *Abstract*

A neural network is useful for generating a solution to problems without having to build underlying principles into the solution beforehand. Here, the pole -balancing problem is studied: a pole begins in a nearly upright position, standing on one end. At each time step, a neural network, which takes as inputs the position, velocity, angular position and angular velocity, must generate a move that keeps the pole upright. It was found that a network which judges error based on deviation from vertical can keep the pole pennanently upright relatively quickly.

## *Background*

### **Simple computing elements -The units of** a **network**

A neural network is called that because it is based on a model of the brain. In the brain, neurons have two states, an excited, or fIring state, and a resting state. Dendrites act as inputs, receiving signals from other neurons, while the axon acts a transmitter, sending out an excitation signal. If enough excitation signals are received from the dendrite, the entire neuron goes into an excited state, and the an electric pulse travels along the axon, where it will meet other dendrites.

Each node in a neural network is similar to the neuron described above (see figure I). Each unit has several input connections Aj and uses an

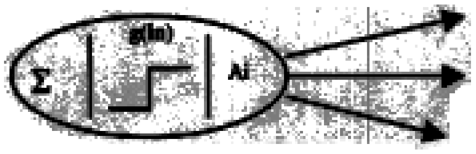
internal function 9 (called the activation function) to send an output signal Ai to other nodes. The activation function 9 takes as input the sum of the products of the input signals and the weights connecting those signals to the input node: in = *L* Aj * Wji. The activation function gives the output value Ai as a function of the input. Typically, the activation function is g(x) = sigmoid(x)
1

= **-This function generates** a **zero or** a **one when** x **is positively or negatively large and gives** a ***l+ex***
**smooth transition between those two values. Continuing the biological analogy,** a **one corresponds to** a **fIring of** a **pulse down the axon and** a **zero corresponds to no firing.**

~
**Input**
**Function**
**A**.. **ctlvatlon**
**Function Figure** 1- A **single node** in a **neural network**

**Input** Connections

Aj



Output
Function

**Output** Connections

The most crucial feature of the neural network is the input weights. These weights, which can be either positive or negative, determine how the input affects the output. Similarly, the brain's behavior is determined by the strength of connections between neurons. It is through adjusting these weights that learning takes place.

Neural Networks and Learning by Example

Network Structure

A network is constructed by linking nodes together in various ways. There are a number of ways to build a neural network, which fall into two basic categories: feed- forward and recurrent networks. Feed - forward networks have a start and end and have no cycles. Recurrent networks have outputs that become inputs to the same network. Although recurrent networks are probably closer to biology, they are more complex and are not used here.

~

Inputs

Hidden Layer

Output(s)

D~O, **D40..**

**Figure** 2- A **simple two layer, feed forward network**

C)

Among feed-forward networks, a simple multilayer network (Figure 2), with inputs, hidden units, and outputs, is commonly used. Like in this example, the model used in these experiments is comprised of two layers. Inputs are angular position and velocity and outputs are signals indicating motion right and left.

Content of Examples

A feed-forward network 'learns' by repetition of examples. Each example consists of a fully defined set of input values and output values corresponding to those inputs. The neural network is initialized with a set of generally arbitrary weights and the network is run on each example to create a set of output values. An error for each example is the difference between generated and actual outputs. Based on the error at the output, values of various weights are either increased or decreased.

If there are a finite number of examples (and there usually are) it is possible to over-specify the sample set. When this is the case, the network performs poorly on an independent set of examples. In order to reduce the likelihood of over-specification, the number of internal nodes must be limited (depending on the number of examples).

Adjustment of the Weights and Backpropagation

**Given an error Errj** in **the output nodes, the weight between output node** i **and node** j **is adjusted with each example according to the following rule:**

*w*

1,1

*=w,*

*+ax Err; xg'(in;)*

Here, g' is the derivative of the sigmoid function described earlier, and alpha is a constant corresponding to the rate of learning. A larger alpha produces faster convergence to a solution, but tends to be less stable. A smaller alpha results in slower convergence of the weights but greater stability.

Backpropagation is a way ofusing the error at the output nodes to adjust weights at edges which are not directly connected to the outputs, but are separated by one or more layers. Backprogation is discussed in greater detail on pp. 579- 581 of Artificial Intelligence. A Modem ADDroach (Russell, et. al.). Essentially, backpropagation generates an error value for layers feeding into the outputs, based on weights. Then, these error values are propagated backward another layer and so on.

**Reinforcement Learning -An Alternative to Backpropagation**

The method of learning described above will be briefly contrasted to reinforcement learning. Whereas learning by example uses error to adjust weights, reinforcement learning uses positive reinforcement by giving a positive value to some goal state. States that are reachable from the goal state are also given a positive value of lesser magnitude, based on likelihood of reaching the goal state. This positive reinforcement is contrasted with the negative reinforcement used in these experiments.

3-

Pole Balancing Description

The constraints on the problem are is follows

A cylindrical pole, of negligible thickness is approximately (but not exactly) upright. It balances on a pivot located at its base that can be moved from side to side. The base either moves discrete amount right or left or stays stationary in each time step. The goal of the neural network is to determine which of these three moves should be performed at each time step to keep the pole upright.

For the sake of simplicity, the pole's motion is entirely planar. The case of motion in three dimensions is not very different from motion in two, because both horizontal directions are essentially equivalent.

Some versions of this problem have the additional constraint that the horizontal position of the pole must remain with a given set of bounds, but that is not a limitation here.

-4

## Introduction

### Physical Model

To keep the model simple, position is specified by just four parameters

- x – position of the base
- linear velocity of the base in x
- angle of the pole
- angular velocity of the pole

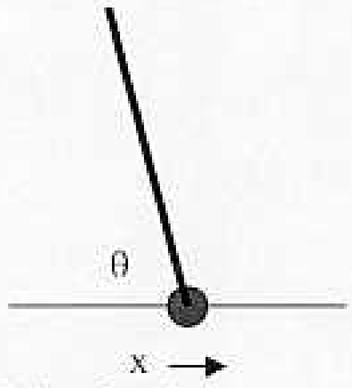


Figure 3 – Pole configuration

Each time step is broken down into ten smaller sub-steps. During each time step one of two things can happen:

(a) The network specifies no motion. In this case, just the force of gravity influences velocities at each time step and velocities are used to compute new positions for each of the ten sub-steps.
(b) The network specifies motion. In this case, the base, which is at rest initially, accelerates uniformly for five sub-steps in the direction specified and then decelerates uniformly during the remaining five sub-steps. The forces from pole movement and gravity are added together in each step.

The pole is initialized to a random angle near vertical and released with no horizontal or angular velocity. The model keeps going until the pole is out of the range of –pi /2 to pi/2. Then the pole is re-initialized at another random angle.

**Reason for Discreteness**

Clearly allowing continuous movements would allow a situation where the pole is placed in a perfectly upright position by very tiny and precise movements. We do not want to allow this. Further, the way a human would do it is through discrete movements.

### Network Model

**Input Parameters**
Angular Position and angular velocity

**Output Parameters**
The outputs of two nodes (move right, move left) are used to generate a move (left, right, stationary

## *Methods*

Equations of motion for the pole

Equations for Gravity

Gravity is computed based on a fixed base

**Moment** of inertia **about end** of rigid **rod** I = **(1/3)** m L 2

T = **mg sin** e r = I a **where** a **is angular acceleration**

**Substituting** r = h / **2, we have**

a = **(3** 9 **sin** e **)/(2h)**

Equations for End Force

End force is computed by considering the pole as a free mass.

By geometry, it is clear that

a = Ax **cos e/r where** Ax **is linear acceleration of the base**

**Again substituting for** r, **we get**

a = 2 Ax **cos e/h**

At each time step, these two effects are added to get the total change in angular motion of the pole:

**~t** + **dt)** = **~t)** + **a(t)** x **dt**

**and e(t** + **dt)** = **e(t)** + **ro(t)** x **dt**

Horizontal motion is simply governed by

**1/(1+d1)** = v **(1)** + **Ax(1)** x **d1 and s(1+d1)** = **s(1)** + v **(1)** x **d1**

## Penalties

Two methods were used to assign error values in each example. The ability of the network to learn in each case was noted.
(a) Assign an error of zero at the output nodes if the pole remained upright. Assign an error of one at the output nodes if the pole fell during the last time step.
(b) Assign an error at every time step as the angle of the pole after the movement. This means that unless the pole is exactly vertical, some learning will take place. It also means that the network can become better at balancing the pole even when the pole does not fall.

6-

Convergence

In each test a convergence of the neural network was considered to have occurred when the network completed 500 consecutive steps without a fall. The number 500 is more or less arbitrary, but it is large enough for our purposes.

Variation of Different Parameters

Height of Pole

The height of the pole was varied between 0.2 meters and 15 meters, and the number of steps before convergence was noted.

**Learning Rate** a

Learning rate alpha was varied between 0.1 and 30, and the number of steps before convergence was noted.

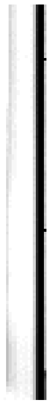
Tweaking the physical model

**In** order to make the task achievable but somewhat challenging for the network, several parameters were adjusted. These parameters were:

*Height olpole;* 1 meter
*Size oltime step;* 1/10 ofa second (each sub-time step is 0.01 second) *Average velocity during pole end movement;* 0.5 meters / second

## Software / Hardware Specifications

The experiment was designed as an applet in Java using Microsoft Visual studio. In has been tested on several different PC -compatible systems in both Internet Explorer and Netscape. Screen sizes of 15 and 17 inches displayed well, and processors running at 166 MHz and better showed the simulation at a good rate. Slower processors might not handle the applet as well.



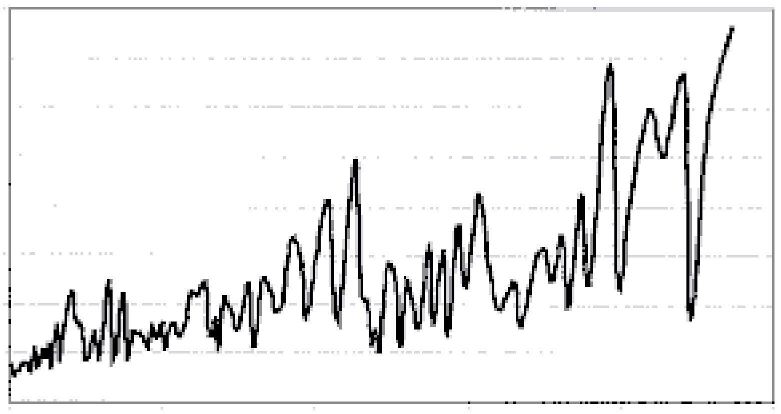
*Results*

## **Convergence to** a **solution**

Shown in figure 4 is a plot of the duration that the pole remains upright as a function of the number of examples that the network has seen. As the figure shows, performance is rather erratic, but generally improving, especially at the beginning.

~ **160**
~ **140 -**
c **120** ~
(1) **100** 0.

S **80** (/)
'0 **60** Q) **40**
.0
E **20** ~
z 0



0

1000

2000

3000

4000

5000

Number of Steps Connpleted

**Figure** 4- **Number of steps until collapse as** a **function of number of steps completed** (examples )

Observation of Learning Rates

In figure 5, learning time is shown as a function of learning rate alpha. Large alpha consistently improves learning rate. Figure 6 shows learning time as a function of height of the pole. An optimum height is around 4 meters, with limits on what can be balanced at both the upper and lower ends.

cn 0.. 0) - cn - O **I.. 0;>** .0 **E: =1 2:**

100000

10000

1000

100

10

~

0.1

1

10

alpha

-8 -

100

**Figure** 5- **Number of steps to convergence as** a **function** of learning **rate alpha Same, with different pole lengths**

**120000** ~ 1 **00000** 00 **80000 -**
~ **60000**
~ **40000** ~ **20000**
0

1--

0

5

10

15

20

Heig ht

**Figure** 6 **-Number of steps to convergence as** a **function of length of pole**

-9

*Conclusions*

The goal with this sort of problem is to achieve a neural network that behaves well *without any pre- programmed physical knowledge* of the system. Indeed this was achieved.

**In** this case, the correlation between alpha and success at learning was purely positive. Perhaps a more complicated network would have given a different result.

The relationship between pole length and learning rate was an interesting one. An optimum length was about four meters. Longer poles took much longer for the network to learn, largely because a lot of base movement is necessary for a small change in angle. Very short poles could not be balanced at all, probably because movements were too coarse.

When modeling applying errors using the two methods described above, a major observation was made. Method (a), whereby error is non-zero only when the pole falls, is insufficient for the network to achieve balance after any finite amount of time. This is because the pole may fall at some time step as a **result of an earlier error.** The most recent move may have been correct, but an earlier move may have been incorrect. Unless a method is built in which propagates error backward through time steps, the network will continue to make bad moves that do not result in a fall in the very next time step. Method (b ), where error is proportional to angle, avoided this problem and proved much more successful.

## *Further experiments*

Working with less information

One way that has been suggested to make the pole -balancing problem more challenging (Jervis, et. al.) is to reduce amount of information available to the network. For example, instead of allowing the network to know precise values of angle and angular velocity, why not connect the network to the outputs of a video camera, so that the network must derive the necessary information from visual cues. This would be a much more interesting problem because there is no simple transformation between the *appearance* of a pole in a given position and the move that would be required.

Propagate backward through several time steps

The present model had to be adjusted so that error did not correspond to falls but to angle at every time step. But a memory of recent moves was kept, then the network could adjust according to those examples with the knowledge that the pole is going to fall in several steps. It would be much more elegant to correct to network only after a fall anyway.

## *References*

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

**Also available online at:**
***http*** : ! *!enry. cs.* ***umass. edu!-rich!booklthe-book. htm.***

**Russell, S. J. and Norvig,** P **.Artificial Intelligence:** A **Modem AI1I1rQach. Prentice Hall, Upper Saddle River, NJ, 1995.**

**Thomton, S.** T **.and Marion,** J **.B. Classic Dvnarnics of Particles and Systems. Harcourt Brace, New** York, **1995.**

Jervis, T.T. and Fallside, F. "Pole Balancing on Real Rig Using a Reinforcement Learning Controller. Cambridge University Engineering Dept., Dec. 16, 1992.